pyRevit Documentation

Release 4.5

eirannejad

Jul 30, 2018

Getting Started

Note: This documentation is a work-in-progress. Thanks for your patience.

Getting Started

I suggest reading this section completely as it provides 99% of what you will need to know for developing scripts in pyRevit environment. Other sections dive deeper into pyRevit inner workings.

CHAPTER 1

Anatomy of a pyRevit Script

pyRevit provides a few basic services to python scripts that use its engine. These fuctionalities are accessible through a few high level modules. This is a quick look at these services and their associated python modules.

1.1 Basic script parameters

- **Command Tooltips**

  Tooltips are shown similarly to the other buttons in Revit interface. You can define the tooltip for a script using the doscstring at the top of the script or by explicitly defining __doc__ parameter.

```
"""You can place the docstring (tooltip) at the top of the script file.
This serves both as python docstring and also button tooltip in pyRevit.
You should use triple quotes for standard python docstrings."""
```

```
__doc__ = 'This is the text for the button tooltip associated with this␣
↪script.'
```

- **Custom Command Title**

  When using the bundle name is not desired or you want to add a newline character to the command name for better display inside Revit UI Panel, you can define the __title__ variable in your script and set it to the desired button title.

```
__title__ = 'Sample\\nCommand'
```

- **Command Author**

  You can define the script author as shown below. This will show up on the button tooltip.

```
__author__ = 'Ehsan Iran-Nejad'
```

1.2 Command Availability

Revit commands use standard `IExternalCommandAvailability` class to let Revit know if they are available in different contexts. For example, if a command needs to work on a set of elements, it can tell Revit to deactivate the button unless the user has selected one or more elements.

In pyRevit, command availability is set through the `__context__` variable. Currently, pyRevit support three types of command availability types.

```
# Tool activates when at least one element is selected
__context__ = 'Selection'

# Tools are active even when there are no documents available/open in Revit
__context__ = 'zerodoc'

# Tool activates when all selected elements are of the given category or categories
__context__ = '<Element Category>'
__context__ = ['<Element Category>', '<Element Category>']
```

- **Element Categories**

  `<Element Category>` can be any of the standard Revit element categories. See *Appendix B: System Category Names* for a full list of system categories. You can use the `List` tool under `pyRevit > Spy` and list the standard categories.

  Here are a few examples:

  ```
  # Tool activates when all selected elements are of the given category

  __context__ = 'Doors'
  __context__ = 'Walls'
  __context__ = 'Floors'
  __context__ = ['Space Tags', 'Spaces']
  ```

1.3 pyrevit.script Module

All pyRevit scripts should use the *pyrevit.script* module to access pyRevit functionality unless listed otherwise. pyRevit internals are subject to changes and accessing them directly is not suggested.

Here is a list of supported modules for pyRevit scripts. Examples of using the functionality in these modules are provided on this page.

pyrevit.script

This module provides access to output window (*pyrevit.output*), logging (`pyrevit.coreutils.logger`), temporary files (`pyrevit.coreutils.appdata`), and other misc features. See the module page for usage examples and full documentation of all available functions.

1.3.1 Logging

You can get the default logger for the script using *pyrevit.script.get_logger()*.

```
from pyrevt import script

logger = script.get_logger()
```

(continues on next page)

(continued from previous page)

```
logger.info('Test Log Level :ok_hand_sign:')

logger.warning('Test Log Level')

logger.critical('Test Log Level')
```

Critical and warning messages are printed in color for clarity. Normally debug messages are not printed. you can hold CTRL and click on a command button to put that command in DEBUG mode and see all its debug messages

```
logger.debug('Yesss! Here is the debug message')
```

1.3.2 Controlling Output Window

Each script can control its own output window:

```
from pyrevit import script

output = script.get_output()

output.set_height(600)
output.get_title()
output.set_title('More control please!')
```

See *Effective Output* for more info.

1.3.3 Script Config

Each script can save and load configuration pyRevit's user configuration file:

See *pyrevit.output* for more examples.

See *pyrevit.script.get_config()* and *pyrevit.script.save_config()* for the individual functions used here.

```
from pyrevit import script

config = script.get_config()

# set a new config parameter: firstparam
config.firstparam = True

# saving configurations
script.save_config()

# read the config parameter value
if config.firstparam:
    do_task_A()
```

1.3.4 Logging Results

pyRevit has a usage logging system that can record all tool usages to either a json file or to a web server. Scripts can return custom data to this logging system.

In example below, the script reports the amount of time it saved to the logging system:

```
from pyrevit import script

results = script.get_results()
results.timesaved = 10
```

1.3.5 Using Temporary Files

Scripts can create 3 different types of data files:

- **Universal files**

  These files are not marked by host Revit version and could be shared between all Revit versions and instances. These data files are saved in pyRevit's appdata directory and are NOT cleaned up at Revit restarts.

  See *pyrevit.script.get_universal_data_file()*

  Note: Script should take care of cleaning up these data files.

  ```
  # provide a unique file id and file extension
  # Method will return full path of the data file
  from pyrevit import script
  script.get_universal_data_file(file_id, file_ext)
  ```

- **Data files**

  These files are marked by host Revit version and could be shared between instances of host Revit version Data files are saved in pyRevit's appdata directory and are NOT cleaned up when Revit restarts.

  See *pyrevit.script.get_data_file()*

  Note: Script should take care of cleaning up these data files.

  ```
  # provide a unique file id and file extension
  # Method will return full path of the data file
  from pyrevit import script
  script.get_data_file(file_id, file_ext)
  ```

- **Instance Data files**

  These files are marked by host Revit version and process Id and are only available to current Revit instance. This avoids any conflicts between similar scripts running under two or more Revit instances. Data files are saved in pyRevit's appdata directory (with extension *.tmp*) and ARE cleaned up when Revit restarts.

  See *pyrevit.script.get_instance_data_file()*

  ```
  # provide a unique file id and file extension
  # Method will return full path of the data file
  from pyrevit import script
  script.get_instance_data_file(file_id)
  ```

- **Document Data files**

  (Shared only between instances of host Revit version): These files are marked by host Revit version and name of Active Project and could be shared between instances of host Revit version. Data files are saved in pyRevit's appdata directory and are NOT cleaned up when Revit restarts.

  See *pyrevit.script.get_document_data_file()*

  Note: Script should take care of cleaning up these data files.

```
# provide a unique file id and file extension
# Method will return full path of the data file
from pyrevit import script
script.get_document_data_file(file_id, file_ext)

# You can also pass a document object to get a data file for that
# document (use document name in file naming)
script.get_document_data_file(file_id, file_ext, doc)
```

1.4 Appendix A: Builtin Parameters Provided by pyRevit Engine

Variables listed below are provided for every script in pyRevit.

Note: It's strongly advised not to read or write values from these variables unless necessary. The *pyrevit* module provides wrappers around these variables that are safe to use.

```
# Revit UIApplication is accessible through:
__revit__

# Command data provided to this command by Revit is accessible through:
__commandData__

# selection of elements provided to this command by Revit
__elements__

# pyRevit engine manager that is managing this engine
__ipyenginemanager__

# This variable is True if command is being run in a cached engine
__cachedengine__

# pyRevit external command object wrapping the command being run
__externalcommand__

# information about the pyrevit command being run
__commandpath__              # main script path
__alternatecommandpath__     # alternate script path
__commandname__              # command name
__commandbundle__            # command bundle name
__commandextension__         # command extension name
__commanduniqueid__          # command unique id
```

(continues on next page)

(continued from previous page)

```
# This variable is True if user CTRL-Clicks the button
__forceddebugmode__

# This variable is True if user SHIFT-Clicks the button
__shiftclick__

# results dictionary
__result__
```

1.5 Appendix B: System Category Names

```
Adaptive Points
Air Terminal Tags
Air Terminals
Analysis Display Style
Analysis Results
Analytical Beam Tags
Analytical Beams
Analytical Brace Tags
Analytical Braces
Analytical Column Tags
Analytical Columns
Analytical Floor Tags
Analytical Floors
Analytical Foundation Slabs
Analytical Isolated Foundation Tags
Analytical Isolated Foundations
Analytical Link Tags
Analytical Links
Analytical Node Tags
Analytical Nodes
Analytical Slab Foundation Tags
Analytical Spaces
Analytical Surfaces
Analytical Wall Foundation Tags
Analytical Wall Foundations
Analytical Wall Tags
Analytical Walls
Annotation Crop Boundary
Area Load Tags
Area Tags
Areas
Assemblies
Assembly Tags
Boundary Conditions
Brace in Plan View Symbols
Cable Tray Fitting Tags
Cable Tray Fittings
Cable Tray Runs
Cable Tray Tags
Cable Trays
Callout Boundary
Callout Heads
Callouts
```

(continues on next page)

(continued from previous page)

```
Cameras
Casework
Casework Tags
Ceiling Tags
Ceilings
Color Fill Legends
Columns
Communication Device Tags
Communication Devices
Conduit Fitting Tags
Conduit Fittings
Conduit Runs
Conduit Tags
Conduits
Connection Symbols
Contour Labels
Crop Boundaries
Curtain Grids
Curtain Panel Tags
Curtain Panels
Curtain System Tags
Curtain Systems
Curtain Wall Mullions
Data Device Tags
Data Devices
Detail Item Tags
Detail Items
Dimensions
Displacement Path
Door Tags
Doors
Duct Accessories
Duct Accessory Tags
Duct Color Fill
Duct Color Fill Legends
Duct Fitting Tags
Duct Fittings
Duct Insulation Tags
Duct Insulations
Duct Lining Tags
Duct Linings
Duct Placeholders
Duct Systems
Duct Tags
Ducts
Electrical Circuits
Electrical Equipment
Electrical Equipment Tags
Electrical Fixture Tags
Electrical Fixtures
Electrical Spare/Space Circuits
Elevation Marks
Elevations
Entourage
Filled region
Fire Alarm Device Tags
Fire Alarm Devices
```

(continues on next page)

(continued from previous page)

```
Flex Duct Tags
Flex Ducts
Flex Pipe Tags
Flex Pipes
Floor Tags
Floors
Foundation Span Direction Symbol
Furniture
Furniture System Tags
Furniture Systems
Furniture Tags
Generic Annotations
Generic Model Tags
Generic Models
Grid Heads
Grids
Guide Grid
HVAC Zones
Imports in Families
Internal Area Load Tags
Internal Line Load Tags
Internal Point Load Tags
Keynote Tags
Level Heads
Levels
Lighting Device Tags
Lighting Devices
Lighting Fixture Tags
Lighting Fixtures
Line Load Tags
Lines
Masking Region
Mass
Mass Floor Tags
Mass Tags
Matchline
Material Tags
Materials
Mechanical Equipment
Mechanical Equipment Tags
MEP Fabrication Containment
MEP Fabrication Containment Tags
MEP Fabrication Ductwork
MEP Fabrication Ductwork Tags
MEP Fabrication Hanger Tags
MEP Fabrication Hangers
MEP Fabrication Pipework
MEP Fabrication Pipework Tags
Multi-Category Tags
Multi-Rebar Annotations
Nurse Call Device Tags
Nurse Call Devices
Panel Schedule Graphics
Parking
Parking Tags
Part Tags
Parts
```

(continues on next page)

(continued from previous page)

```
Pipe Accessories
Pipe Accessory Tags
Pipe Color Fill
Pipe Color Fill Legends
Pipe Fitting Tags
Pipe Fittings
Pipe Insulation Tags
Pipe Insulations
Pipe Placeholders
Pipe Segments
Pipe Tags
Pipes
Piping Systems
Plan Region
Planting
Planting Tags
Plumbing Fixture Tags
Plumbing Fixtures
Point Clouds
Point Load Tags
Project Information
Property Line Segment Tags
Property Tags
Railing Tags
Railings
Ramps
Raster Images
Rebar Cover References
Rebar Set Toggle
Rebar Shape
Reference Lines
Reference Planes
Reference Points
Render Regions
Revision Cloud Tags
Revision Clouds
Roads
Roof Tags
Roofs
Room Tags
Rooms
Routing Preferences
Schedule Graphics
Scope Boxes
Section Boxes
Section Line
Section Marks
Sections
Security Device Tags
Security Devices
Shaft Openings
Sheets
Site
Site Tags
Space Tags
Spaces
Span Direction Symbol
```

(continues on next page)

(continued from previous page)

```
Specialty Equipment
Specialty Equipment Tags
Spot Coordinates
Spot Elevation Symbols
Spot Elevations
Spot Slopes
Sprinkler Tags
Sprinklers
Stair Landing Tags
Stair Paths
Stair Run Tags
Stair Support Tags
Stair Tags
Stair Tread/Riser Numbers
Stairs
Structural Annotations
Structural Area Reinforcement
Structural Area Reinforcement Symbols
Structural Area Reinforcement Tags
Structural Beam System Tags
Structural Beam Systems
Structural Column Tags
Structural Columns
Structural Connection Tags
Structural Connections
Structural Fabric Areas
Structural Fabric Reinforcement
Structural Fabric Reinforcement Symbols
Structural Fabric Reinforcement Tags
Structural Foundation Tags
Structural Foundations
Structural Framing
Structural Framing Tags
Structural Internal Loads
Structural Load Cases
Structural Loads
Structural Path Reinforcement
Structural Path Reinforcement Symbols
Structural Path Reinforcement Tags
Structural Rebar
Structural Rebar Coupler Tags
Structural Rebar Couplers
Structural Rebar Tags
Structural Stiffener Tags
Structural Stiffeners
Structural Truss Tags
Structural Trusses
Switch System
Telephone Device Tags
Telephone Devices
Text Notes
Title Blocks
Topography
View Reference
View Titles
Viewports
Views
```

(continues on next page)

(continued from previous page)

```
Wall Tags
Walls
Window Tags
Windows
Wire Tags
Wires
Zone Tags
```

CHAPTER 2

Effective Output

2.1 Clickable Element Links

work in progress

2.2 Progress bars

work in progress

2.3 Standard Forms

work in progress

2.4 Graphs

Step 1: Create a chart object for the chart type that you want. We'll add data to this later...

```
from pyrevit import script

output = script.get_output()

# Line chart
chart = output.make_line_chart()
# Bar chart
chart = output.make_bar_chart()
# Bubble chart
chart = output.make_bubble_chart()
```

(continues on next page)

(continued from previous page)

```
# Radar chart
chart = output.make_radar_chart()
# Polar chart
chart = output.make_polar_chart()
# Pie chart
chart = output.make_pie_chart()
# Doughnut chart
chart = output.make_doughnut_chart()
```

Step 1-a: Optional: Setup the chart title, and other options. the full list of options for every chart is available on Charts.js Documentation page. Some of the properties have their own sub-properties, for example the `title` option for the charts has multiple sub-properties as shown below. The value for these type of properties should be a dictionary of the sub-properties you'd like to set. All this is explained clearly in the Charts.js Documentation

```
chart.set_style('height:150px')

chart.options.title = {'display': True,
                       'text':'Chart Title',
                       'fontSize': 18,
                       'fontColor': '#000',
                       'fontStyle': 'bold'}
```

Step 2: Now let's add data to the chart. Every chart object has a data property `chart.data` that we can interact with to add datasets to the chart. Different types of charts need different types of data sets in terms of how data is organized, so the chart can present multiple data sets correctly. I'm providing two examples here, one for a simple line chart (showing 3 different data sets) and another for a radial chart (also showing 3 different data sets within the same chart). They're all very similar to each other though.

2.4.1 Line charts

See the comments in the script for more info

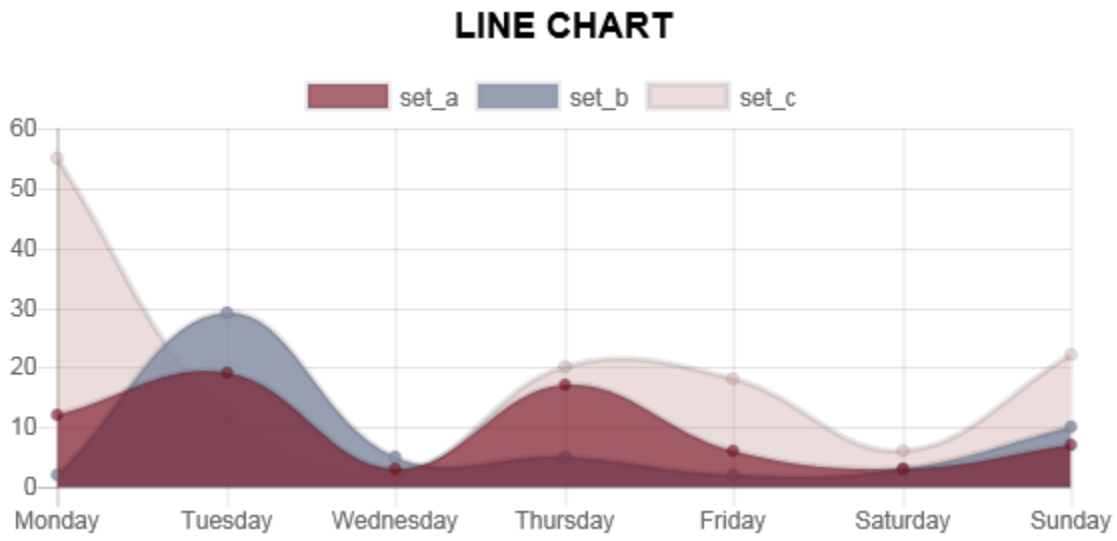
```
# this is a list of labels for the X axis of the line graph
chart.data.labels = ['Monday', 'Tuesday',
                     'Wednesday', 'Thursday',
                     'Friday', 'Saturday', 'Sunday']

# Let's add the first dataset to the chart object
# we'll give it a name: set_a
set_a = chart.data.new_dataset('set_a')
# And let's add data to it.
# These are the data for the Y axis of the graph
# The data length should match the length of data for the X axis
set_a.data = [12, 19, 3, 17, 6, 3, 7]
# Set the color for this graph
set_a.set_color(0xFF, 0x8C, 0x8D, 0.8)
# You can also set custom options for this graph
# See the Charts.js documentation for all the options
set_b.fill = False

# Same as above for a new data set: set_b
set_b = chart.data.new_dataset('set_b')
# Obviously a different set of data and a different color
set_b.data = [2, 29, 5, 5, 2, 3, 10]
set_b.set_color(0xFF, 0xCE, 0x56, 0.8)
```

(continues on next page)

(continued from previous page)

```
# Same as above for a new data set: set_c
set_c = chart.data.new_dataset('set_c')
# Obviously a different set of data and a different colorset_c.data = [55, 12, 2, 20,␣
↪18, 6, 22]
set_c.set_color(0x36, 0xA2, 0xEB, 0.8)
```

And here is the result:



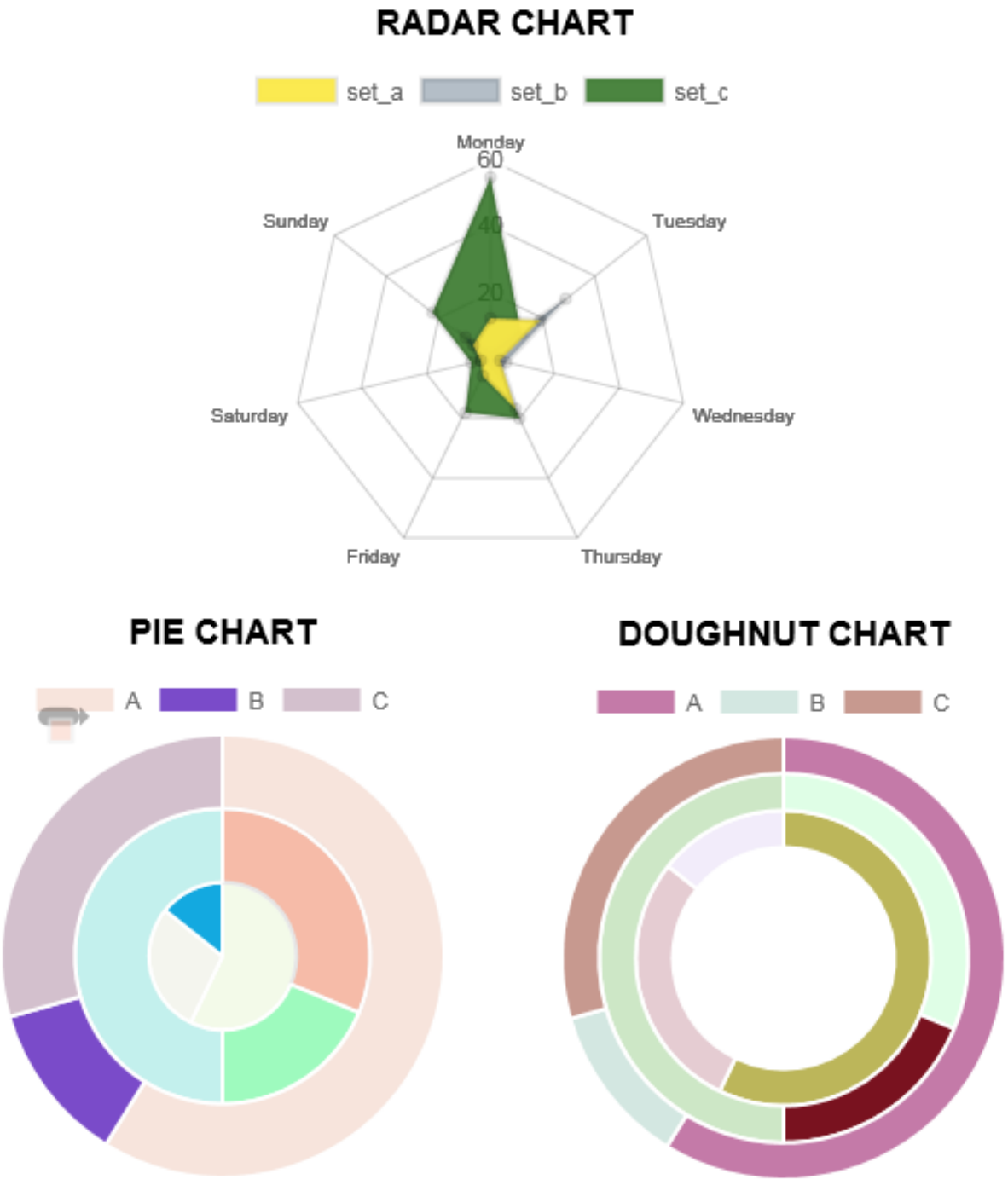
2.4.2 Pie charts

See the comments in the script for more info

```
# Set the labels for the circumference axis
chart.data.labels = ['A', 'B', 'C']

# Create new data sets
set_a = chart.data.new_dataset('set_a')
set_a.data = [100, 20, 50]
# You can set a different color for each pie of the chart
set_a.backgroundColor = ["#560764", "#1F6CB0", "#F98B60"]

set_b = chart.data.new_dataset('set_b')
set_b.data = [50, 30, 80]
set_b.backgroundColor = ["#913175", "#70A3C4", "#FFC057"]

set_c = chart.data.new_dataset('set_c')
set_c.data = [40, 20, 10]
set_c.backgroundColor = ["#DD5B82", "#E7E8F5", "#FFE084"]
```

2.4.3 Other charts

You can apply these data sets for radar, pie, polar, and doughnut (since they're all radial) charts and get different results:



Step 3: The last step is to ask the chart object to draw itself.

```
# Before drawing the chart you can randomize the colors
# if you have not added any color to the datasets.
chart.randomize_colors()

# Finally let's draw the chart
chart.draw()
```

2.4.4 Charts engine

Here is a little info on how the charts engine work: the pyRevit charts module is `pyrevit.coreutils.charts`. This is the module that the output window interacts with to create the charts.

The charts module provides the chart object and handles the creation of datasets. The first thing it does when drawing the graph is to create a html `<canvas>` element and assign a unique id to it:

```
<canvas id="chart123456"></canvas>
```

Then it parses the input data and create a JSON representation of the data. The JSON string (*json_data*) will be inserted into a template javascript. This javascript is in turn, creating a Chart object from the `Chart.js` library:

```
var ctx = document.getElementById('{}').getContext('2d');
var chart = new Chart(ctx, json_data);
```

and finally, the pyRevit chart object, injects this dynamically created javascript into the <*head*> of the output window WebBrowser component:

```
output.inject_script(js_code)
```

CHAPTER 3

Keyboard Shortcuts

3.1 Shift-Click: Alternate/Config Script

Each pyRevit command bundle can contain two scripts:

- `*script.py` is the main script.
- `*config.py` is the Alternale/Config script.

SHIFT-clicking on a ui button will run the alternate/config script. This alternate script is generally used to configure the main tool. Try Shift clicking on the Match tool in pyRevit > Modify panel and see the configuration window. Then try Shift clicking on the Settings tool in pyRevit panel slide-out and see what it does.

If you don't define the configuration script, you can check the value of `__shiftclick__` in your scripts to change script behaviour. This is the method that the Settings command is using to open the config file location in explorer:

```
if __shiftclick__:
    do_task_A()
else:
    do_task_B()
```

3.2 Ctrl-Click: Debug Mode

CTRL-clicking on a ui button will run the script in DEBUG mode and will allow the script to print all debug messages. You can check the value of `__forceddebugmode__` variable to see if the script is running in Debug mode to change script behaviour if neccessary.

```
if __forceddebugmode__:
    do_task_A()
else:
    do_task_B()
```

3.3 Alt-Click: Show Script file in Explorer

ALT-clicking on a ui button will show the associated script file in windows explorer.

3.4 Ctrl-Shift-Alt-Click: Reload Engine

If you're using pyRevit Rocket mode, this keyboard combination will force pyRevit to discard the cached engine for this command and use a new fresh engine. If you are developing scripts for pyRevit and using external modules, you'll need to use this keyboard combination after changes to the imported module source codes. Since the modules are already imported in the cached engine, you'd need a new fresh engine to reload the modules.

3.5 Shift-Win-Click: Show Script file in Explorer

Shows the context menu for the pyRevit command. See image below:

CHAPTER 4

Extensions and Commmands

4.1 Why do I need an Extension

pyRevit's extensions system has evolved again to be more flexible and easier to work with. We'll dive right into how you can add your own extension, but first let's answer one important question:

Q: Why would I need to create a separate extension? Why Can't I just add my scripts to the current pyRevit tools?

A: Because pyRevit is a git repository and the real benefit of that is that you can keep it always updated without the need to unsinstall and install the newer versions. To keep this system running without issues, I highly recommend not to mess with the pyRevit git repository folders and contents and pyRevit makes it really easy to add your own extensions. You can even add tools to the standard pyRevit tab in your own extensions. I'll show you how.

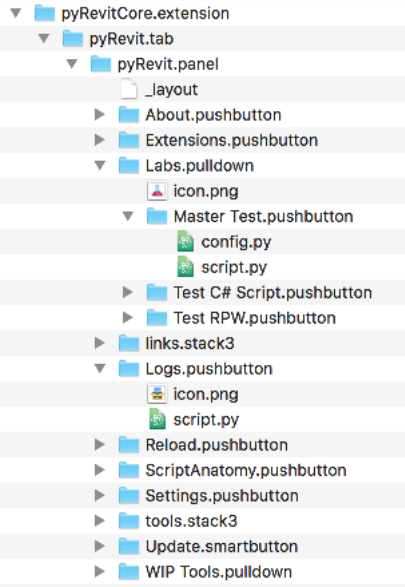
Besides, by creating a separate extension, you'll have all your precious scripts and tools in a safe place and away from the changes being made to the core pyRevit. They can even live somewhere on your company shared drives and be shared between your teams.

4.2 Extensions

Each extension is a group of tools, organized in bundles to be easily accessible through the user interface.

Extensions are organized in a folder bundle with `.extension` postfix.

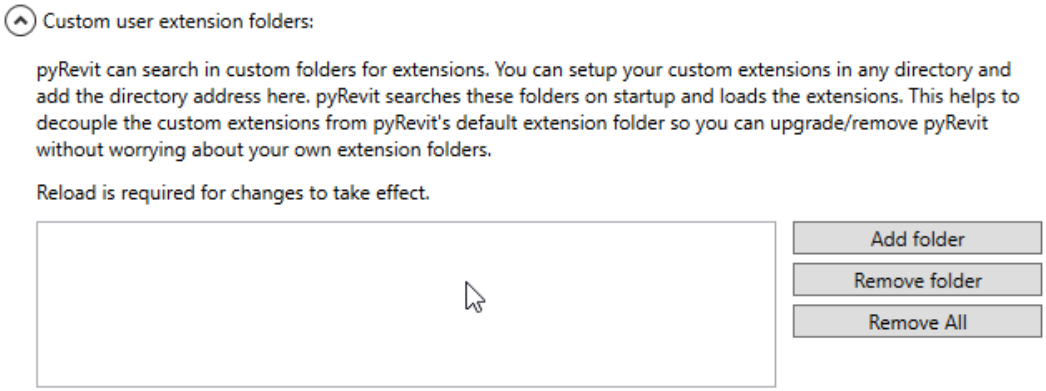
Like this one:



There are two steps that you need to follow to create your own extensions:

- Setup external extension folder:

  First, is to create a separate folder for all your custom extensions and tell pyRevit to load your extensions from this folder. This is done in the Settings window, under the Custom Extension folders section. This way your precious extensions will stay out of the pyRevit installation and are safe.



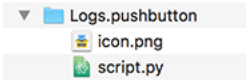
- Create custom extension bundle:

  Next, create your `<your extension name>.extension` folder under your custom extensions folder. Read the sections below on how to create bundles for your commands and the user interface.

4.3 Command Bundles

A bundle is a folder named in the format `bundle_name.bundle_type`.

Like these:



The most basic bundle is a command bundle. There are more than one type of command bundles but a `.pushbutton` bundle explained here covers 90% of the use cases.

4.3.1 Pushbutton Bundle

Each command bundle needs to include a script either in python or C#:

script.py:

> The first script file under the bundle that ends with `script.py` will be used as the script for this command bundle.
>
> Examples: `BuildWall_script.py` `Analyse-script.py`

config.py:

> This for python commands configuration. If this script is provided then Shift-Clicking on the button will run this command instead. Also a black dot will be added to the button name in the user interface to show that this command has a custom configuration tool. See *Shift-Click: Alternate/Config Script*

script.cs:

> This for C# commands and works similarly to python scripts. This C# script will be compiled in runtime.

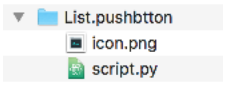
icon.png:

> Command bundles can include an icon for their user interface.

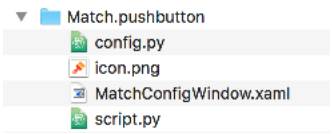
lib/:

> Bundles can define a python library (a sub-folder named *lib* inside the bundle will do). This library will be accessible to the python script in this bundle. This organizes all the python modules that are necessary for this python script to work into one folder.

This is how a command bundle looks like:



And this is a more advanced command bundle with a configuration script and configuration window definition file:



4.4 Group Bundles

Now that we have explained the command bundles, we need a way to organize these commands into a user-friendly interface. Let's introduce **Group Bundles**

A group bundle is a bundle that can contain command bundles and other group bundles. They come in all different shapes and sizes but they have a few features in common:

- They can contain command bundles and other group bundles. (But I've already said that)

- **icon.png**: Bundle can include an icon for their user interface.
- **lib/**: The can define a python library (a sub-folder named `lib` inside the bundle will do). This library will be accessible to all the commands in this bundle and other child group bundles. This folder can contain all the python modules that are being shared between the child commands.
- **_layout**: This is a text file inside the bundle that defines the order in which the bundle contents should be created in the user interface. The contents of this file should be the names of the component in the order that they should be created in the user interface.

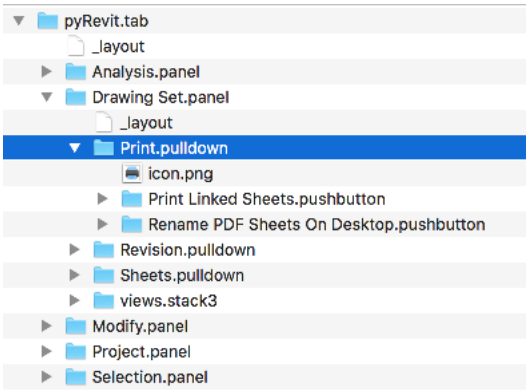
Here is `_layout` file example. This is a layout file for a Group Bundle that has a series of push buttons and other group bundles under itself:

```
PushButton A
PushButton B
PullDown A
---
PullDown B
Stack3 A
>>>
PushButton C
PullDown C
```

Oh, and also:

- `---` This line will add a separator to the interface (You can use more than 3 - characters. For example `----------` still works as a separator)
- `>>>` Any bundle after this line will be created inside a slide-out. This works for panel bundles only. (You can use more than 3 > characters. For example `>>>>>>>>>` still works as a slide-out)

And this is how a typical Group Bundle looks like:



Now let's talk about the different Group Bundles:

4.4.1 Tab Bundle

This bundle creates a Tab in the Ribbon with the bundle name.

Example	**Can Contain**
`pyRevit.tab`	Only `.panel` Group Bundles.

4.4.2 Panel Bundle

This bundle creates a Panel in a Ribbon Tab with the bundle name.

Example	**Can Contain**
`pyRevit.panel`	Any other bundle type

4.4.3 PullDown Bundle

This bundle creates a Pulldown Button in a Ribbon Panel or a Stack, with the bundle name and icon.

Example	**Can Contain**
`pyRevit.pulldown`	Only command bundles

4.4.4 SplitButton Bundle

This bundle creates a Split Button button in a Ribbon Panel or a Stack, with the bundle name and icon.

Example	**Can Contain**
`pyRevit.splitbutton`	Only command bundles

4.4.5 SplitPushButton Bundle

This bundle creates a Split Push Button button (The sticky split button) in a Ribbon Panel or a Stack, with the bundle name and icon.

Example	**Can Contain**
`pyRevit.splitpushbutton`	Only command bundles

4.4.6 Stack Bundle: Two Buttons

This bundle creates a stack of 2 buttons in a panel.

Example	**Can Contain**
`pyRevit.stack2`	Can contain: *.pulldown .splitbutton .splitpushbutton*

4.4.7 Stack Bundle: Three Buttons

Just like the *.stack2* bundle but with 3 buttons instead.

4.5 Advanced Bundles

There are a few more advanced bundle types in pyRevit as well. Here is some quick intro on these bundles.

4.5.1 Smart Button Bundle

Example
`pyRevit.smartbutton`

Smart buttons are python scripts that are written like modules. They should define `__selfinit__` function as shown below. This function gets executed at startup time to give a chance to the button to initialize itself (e.g set its icon based on its state).

The `__selfinit__` must return `True` if the initialization is successful and `False` if it is not. pyRevit will not create the button if the initialization returns `False` and is unsuccessful.

```
def __selfinit__(script_cmp, ui_button_cmp, __rvt__):
    """
    Args:
        script_cmp: script component that contains info on this script
        ui_button_cmp: this is the UI button component
        __rvt__: Revit UIApplication

    Returns:
                    bool: Return True if successful, False if not
    """

    run_self_initialization()
```

4.5.2 No Button Bundle

Example
`pyRevit.nobutton`

No-Button bundles are just like Pushbutton bundles except that they will never show up inside Revit UI and thus don't need any icons. The only method to run these commands is through pyRevit Search tool. These commands are meant for more advanced commands that not every user needs.

4.5.3 Panel Button Bundle

Example
`pyRevit.panelbutton`

Panle Button bundles are just like Pushbutton bundles except that they will be set as the panel configuration button (small arrow at the corner of UI Panels). These bundles do not need to have an icon as the standard Small arrow icon is used for panel configuration buttons by default. These commands work just like any pyRevit command but their primary purpose should be to configure set of related tools in a panel.

4.5.4 Link Button Bundle

Example
`pyRevit.linkbutton`

Link buttons can call a function from another Addin. To make a link button define the parameters below in the bundles `script.py`:

Note: For this button to work properly, the target addin must be already loaded when this button is being created, otherwise Revit can not tie the UI button to an assembly that is not loaded.

```
__assembly__ = 'Addin assembly name'
__commandclass__ = 'Class name for the command'
```

For example to call the Interactive Python Shell from RevitPythonShell addin:

```
__assembly__ = 'RevitPythonShell'
__commandclass__ = 'IronPythonConsoleCommand'
```

4.6 Other Extension Types

4.6.1 Library Extensions

Library extensions are created to share IronPython modules between all extensions. They're in essence IronPython module packages. Some users might decide to develop an IronPython library (e.g. RevitPythonWrapper Library) that other users can use in their tools and benefit from.

Library extensions are identified by `.lib` postfix. The library extension folder address will be added to the `sys.path` of all the other extensions by the loader.

CHAPTER 5

pyRevit Configuration

work in progress

CHAPTER 6

Usage Logger

work in progress

CHAPTER 7

pyRevit Installer

work in progress

pyRevit Core

- *Load Sequence, Step 1: Revit Addon*
- *Load Sequence, Step 2: IronPython Module*

CHAPTER 8

Load Sequence, Step 1: Revit Addon

8.1 The Complex Relationship of a C# Addin and a Python Script

Let's talk basics:

- Revit Addons are written in C# and are windows `.dll` files.
- pyRevit is written as an IronPython module. (actually a bit more complex than that)
- Revit doesn't have an option to run external python scripts.

Thus, we need a way to teach Revit how to run a python script when it's starting up.

The solution was to create a custom C# addin to create a python engine and run a script. We'll call this addin `pyRevitLoader.dll`. I wanted to keep this addin as simple as possible since it's the only statically-compiled piece of code in this project. The rest of the task of loading pyRevit were assigned to a loader python script that is being run by the loader addin.

So:

- `pyRevitLoader.dll` is a simple C# addin for Revit that runs python scripts
- `pyRevitLoader.dll` loads `pyRevitLoader.py` at startup.
- `pyRevitLoader.py` sets up the environment and loads pyRevit.

It's that simple really. See the sources below.

From here on, the documentation page for the *pyrevit.loader* module will take you through all the steps of parsing extensions, making dll assemblies and creating the user interface for the parsed extensions.

8.2 pyRevit loader script

Here is the full source of `pyRevitLoader.py`. The docstring explains how it works.

```
# -*- coding: utf-8 -*-
"""
This is the starting point for pyRevit. At Revit loads the PyRevitLoader.dll
 addon at startup. This dll then creates an ironpython engine and runs
 pyRevitLoader.py (this script). It's the job of this script to setup the
 environment for the pyrevit module (pyrevitlib\pyrevit) and load a new pyRevit
 session. This script needs to add the directory path of the pyrevit lib folder
 so the pyrevit module can be imported and used.
"""

import sys
import os.path as op

# add the library location to the system search paths
sys.path.append(op.dirname(op.dirname(op.dirname(op.dirname(__file__)))))

# now pyrevit can be imported
from pyrevit.loader import sessionmgr

# ask sessionmgr to start a new session
sessionmgr.load_session()
```

8.3 pyRevitLoader Addin Source

The source code for pyRevitLoader addin is under: `pyrevitlib/pyrevit/addin/<loader version>/Source`

CHAPTER 9

Load Sequence, Step 2: IronPython Module

work in progress

Modules

- *pyrevit*
- *pyrevit.api*
- *pyrevit.framework*
- *pyrevit.script*
- *pyrevit.userconfig*
- *pyrevit.output*

CHAPTER 10

pyrevit

10.1 Usage

```
from pyrevit import DB, UI
from pyrevit import PyRevitException, PyRevitIOError

# pyrevit module has global instance of the
# _HostAppPostableCommand and _ExecutorParams classes already created
# import and use them like below
from pyrevit import HOST_APP
from pyrevit import EXEC_PARAMS
```

10.2 Documentation

class pyrevit.**PyRevitException**
Base class for all pyRevit Exceptions.

Parameters args and message are derived from Exception class.

class pyrevit.**PyRevitIOError**
Generic IO error in pyRevit.

class pyrevit.**_HostAppPostableCommand**(*name*, *key*, *id*, *rvtobj*)
Private namedtuple for passing information about a PostableCommand

name
str – Postable command name

key
str – Postable command key string

id
int – Postable command id

rvtobj
RevitCommandId – Postable command Id Object

class pyrevit.**_HostApplication**(*host_uiapp*)
Private Wrapper for Current Instance of Revit.

Provides version info and comparison functionality, alongside providing info on the active screen, active document and ui-document, available postable commands, and other functionality.

Parameters **host_uiapp** (UIApplication) – Instance of running host.

Example

```
>>> hostapp = _HostApplication(__revit__)
>>> hostapp.is_newer_than(2017)
```

activeview
Return view that is active (UIDocument.ActiveView).

app
Return Application provided to the running command.

available_servers
Return list of available Revit server names.

build
str – Return build number (e.g. '20170927_1515(x64)').

doc
Return active Document.

docs
Return list of open Document objects.

get_postable_commands()
Return list of postable commands.

Returns list of *_HostAppPostableCommand*

is_exactly(*version*)
bool: Return True if host app is equal to provided version.

Parameters **version** (*str or int*) – version to check against.

is_newer_than(*version*)
bool: Return True if host app is newer than provided version.

Parameters **version** (*str or int*) – version to check against.

is_older_than(*version*)
bool: Return True if host app is older than provided version.

Parameters **version** (*str or int*) – version to check against.

proc
System.Diagnostics.Process – Return current process object.

proc_id
int – Return current process id.

proc_name
str – Return current process name.

proc_path
str – Return file path for the current process main module.

proc_screen
`intptr` – Return handle to screen hosting current process.

proc_screen_scalefactor
float – Return scaling for screen hosting current process.

proc_screen_workarea
`System.Drawing.Rectangle` – Return screen working area.

uiapp
Return UIApplication provided to the running command.

uidoc
Return active UIDocument.

username
str – Return the username from Revit API (Application.Username).

version
str – Return version number (e.g. '2018').

version_name
str – Return version name (e.g. 'Autodesk Revit 2018').

class `pyrevit.`**_ExecutorParams**
Private Wrapper that provides runtime environment info.

command_data
`ExternalCommandData` – Return current command data.

command_mode
bool – Check if pyrevit is running in pyrevit command context.

command_name
str – Return current command name.

command_path
str – Return current command path.

doc_mode
bool – Check if pyrevit is running by doc generator.

engine_mgr
`PyRevitBaseClasses.EngineManager` – Return engine manager.

engine_ver
str – Return PyRevitLoader.ScriptExecutor hardcoded version.

executed_from_ui
bool – Check if command was executed from ui.

first_load
bool – Check whether pyrevit is not running in pyrevit command.

forced_debug_mode
bool – Check if command is in debug mode.

pyrevit_command
`PyRevitBaseClasses.PyRevitCommandRuntime` – Return command.

result_dict
Dictionary<String, String> – Return results dict for logging.

window_handle
PyRevitBaseClasses.ScriptOutput – Return output window.

10.3 Implementation

```
"""pyRevit root level config for all pyrevit sub-modules."""

import clr
import sys
import os
import os.path as op
from collections import namedtuple
import traceback


try:
    clr.AddReference('PyRevitLoader')
except Exception as e:
    # probably older IronPython engine not being able to
    # resolve to an already loaded assembly.
    # PyRevitLoader is executing this script so it should be referabe.
    pass

try:
    import PyRevitLoader
except ImportError:
    # this means that pyRevit is _not_ being loaded from a pyRevit engine
    # e.g. when importing from RevitPythonShell
    PyRevitLoader = None


PYREVIT_ADDON_NAME = 'pyRevit'
VERSION_MAJOR = 4
VERSION_MINOR = 5
BUILD_METADATA = ''

# ------------------------------------------------------------------------------
# config environment paths
# ------------------------------------------------------------------------------
# main pyrevit repo folder
try:
    # 3 steps back for <home>/Lib/pyrevit
    HOME_DIR = op.dirname(op.dirname(op.dirname(__file__)))
except NameError:
    raise Exception('Critical Error. Can not find home directory.')

# default extensions directory
EXTENSIONS_DEFAULT_DIR = op.join(HOME_DIR, 'extensions')

# main pyrevit lib folders
MAIN_LIB_DIR = op.join(HOME_DIR, 'pyrevitlib')
MISC_LIB_DIR = op.join(HOME_DIR, 'site-packages')
```

(continues on next page)

(continued from previous page)

```
# path to pyrevit module
PYREVIT_MODULE_DIR = op.join(MAIN_LIB_DIR, 'pyrevit')

# loader directory
LOADER_DIR = op.join(PYREVIT_MODULE_DIR, 'loader')

# addin directory
ADDIN_DIR = op.join(LOADER_DIR, 'addin')

# if loader module is available means pyRevit is being executed by Revit.
if PyRevitLoader:
    PYREVITLOADER_DIR = \
        op.join(ADDIN_DIR, PyRevitLoader.ScriptExecutor.EngineVersion)
    ADDIN_RESOURCE_DIR = op.join(PYREVITLOADER_DIR,
                                 'Source', 'pyRevitLoader', 'Resources')
# otherwise it might be under test, or documentation processing.
# so let's keep the symbols but set to None (fake the symbols)
else:
    PYREVITLOADER_DIR = ADDIN_RESOURCE_DIR = None

# add the framework dll path to the search paths
sys.path.append(ADDIN_DIR)
sys.path.append(PYREVITLOADER_DIR)


# pylama:ignore=E402
# now we can start importing stuff
from pyrevit.framework import Process
from pyrevit.framework import Windows
from pyrevit.framework import Forms
from pyrevit.api import DB, UI  # noqa pylama ignore DB not being used here

# ------------------------------------------------------------------------------
# Base Exceptions
# ------------------------------------------------------------------------------
TRACEBACK_TITLE = 'Traceback:'


# General Exceptions
class PyRevitException(Exception):
    """Base class for all pyRevit Exceptions.

    Parameters args and message are derived from Exception class.
    """

    def __str__(self):
        """Process stack trace and prepare report for output window."""
        sys.exc_type, sys.exc_value, sys.exc_traceback = sys.exc_info()
        try:
            tb_report = traceback.format_tb(sys.exc_traceback)[0]
            if self.args:
                message = self.args[0]
                return '{}\n\n{}\n{}'.format(message,
                                             TRACEBACK_TITLE,
                                             tb_report)
            else:
                return '{}\n{}'.format(TRACEBACK_TITLE, tb_report)
```

(continues on next page)

(continued from previous page)

```
        except Exception:
            return Exception.__str__(self)


class PyRevitIOError(PyRevitException):
    """Generic IO error in pyRevit."""

    pass


# ------------------------------------------------------------------------------
# Wrapper for __revit__ builtin parameter set in scope by C# Script Executor
# ------------------------------------------------------------------------------
# namedtuple for passing information about a PostableCommand
_HostAppPostableCommand = namedtuple('_HostAppPostableCommand',
                                     ['name', 'key', 'id', 'rvtobj'])
"""Private namedtuple for passing information about a PostableCommand

Attributes:
    name (str): Postable command name
    key (str): Postable command key string
    id (int): Postable command id
    rvtobj (``RevitCommandId``): Postable command Id Object
"""


class _HostApplication:
    """Private Wrapper for Current Instance of Revit.

    Provides version info and comparison functionality, alongside providing
    info on the active screen, active document and ui-document, available
    postable commands, and other functionality.

    Args:
        host_uiapp (``UIApplication``): Instance of running host.

    Example:
        >>> hostapp = _HostApplication(__revit__)
        >>> hostapp.is_newer_than(2017)
    """

    def __init__(self, host_uiapp):
        self._uiapp = host_uiapp
        self._postable_cmds = []

    @property
    def uiapp(self):
        """Return UIApplication provided to the running command."""
        return self._uiapp

    @property
    def app(self):
        """Return Application provided to the running command."""
        return self.uiapp.Application

    @property
    def uidoc(self):
```

(continues on next page)

(continued from previous page)

```
        """Return active UIDocument."""
        return getattr(self.uiapp, 'ActiveUIDocument', None)

    @property
    def doc(self):
        """Return active Document."""
        return getattr(self.uidoc, 'Document', None)

    @property
    def activeview(self):
        """Return view that is active (UIDocument.ActiveView)."""
        return getattr(self.uidoc, 'ActiveView', None)

    @property
    def docs(self):
        """Return :obj:`list` of open :obj:`Document` objects."""
        return getattr(self.app, 'Documents', None)

    @property
    def available_servers(self):
        """Return :obj:`list` of available Revit server names."""
        return list(self.app.GetRevitServerNetworkHosts())

    @property
    def version(self):
        """str: Return version number (e.g. '2018')."""
        return self.app.VersionNumber

    @property
    def version_name(self):
        """str: Return version name (e.g. 'Autodesk Revit 2018')."""
        return self.app.VersionName

    @property
    def build(self):
        """str: Return build number (e.g. '20170927_1515(x64)')."""
        return self.app.VersionBuild

    @property
    def username(self):
        """str: Return the username from Revit API (Application.Username)."""
        uname = self.app.Username
        uname = uname.split('@')[0]  # if username is email
        # removing dots since username will be used in file naming
        uname = uname.replace('.', '')
        return uname

    @property
    def proc(self):
        """System.Diagnostics.Process: Return current process object."""
        return Process.GetCurrentProcess()

    @property
    def proc_id(self):
        """int: Return current process id."""
        return Process.GetCurrentProcess().Id
```

(continues on next page)

(continued from previous page)

```
    @property
    def proc_name(self):
        """str: Return current process name."""
        return Process.GetCurrentProcess().ProcessName

    @property
    def proc_path(self):
        """str: Return file path for the current process main module."""
        return Process.GetCurrentProcess().MainModule.FileName

    @property
    def proc_screen(self):
        """``intptr``: Return handle to screen hosting current process."""
        return Forms.Screen.FromHandle(
            Process.GetCurrentProcess().MainWindowHandle)

    @property
    def proc_screen_workarea(self):
        """``System.Drawing.Rectangle``: Return screen working area."""
        screen = HOST_APP.proc_screen
        if screen:
            return screen.WorkingArea

    @property
    def proc_screen_scalefactor(self):
        """float: Return scaling for screen hosting current process."""
        screen = HOST_APP.proc_screen
        if screen:
            actual_wdith = Windows.SystemParameters.PrimaryScreenWidth
            scaled_width = screen.PrimaryScreen.WorkingArea.Width
            return abs(scaled_width / actual_wdith)

    def is_newer_than(self, version):
        """bool: Return True if host app is newer than provided version.

        Args:
            version (str or int): version to check against.
        """
        return int(self.version) > int(version)

    def is_older_than(self, version):
        """bool: Return True if host app is older than provided version.

        Args:
            version (str or int): version to check against.
        """
        return int(self.version) < int(version)

    def is_exactly(self, version):
        """bool: Return True if host app is equal to provided version.

        Args:
            version (str or int): version to check against.
        """
        return int(self.version) == int(version)

    def get_postable_commands(self):
```

(continues on next page)

(continued from previous page)

```
        """Return list of postable commands.

        Returns:
            :obj:`list` of :obj:`_HostAppPostableCommand`
        """
        # if list of postable commands is _not_ already created
        # make the list and store in instance parameter
        if not self._postable_cmds:
            for pc in UI.PostableCommand.GetValues(UI.PostableCommand):
                try:
                    rcid = UI.RevitCommandId.LookupPostableCommandId(pc)
                    self._postable_cmds.append(
                        # wrap postable command info in custom namedtuple
                        _HostAppPostableCommand(name=str(pc),
                                                key=rcid.Name,
                                                id=rcid.Id,
                                                rvtobj=rcid)
                        )
                except Exception:
                    # if any error occured when querying postable command
                    # or its info, pass silently
                    pass

        return self._postable_cmds


try:
    # Create an intance of host application wrapper
    # making sure __revit__ is available
    HOST_APP = _HostApplication(__revit__)  # noqa
except Exception:
    raise Exception('Critical Error: Host software is not supported. '
                    '(__revit__ handle is not available)')


# ------------------------------------------------------------------------------
# Wrapper to access builtin parameters set in scope by C# Script Executor
# ------------------------------------------------------------------------------
class _ExecutorParams(object):
    """Private Wrapper that provides runtime environment info."""

    @property   # read-only
    def engine_mgr(self):
        """``PyRevitBaseClasses.EngineManager``: Return engine manager."""
        try:
            return __ipyenginemanager__
        except NameError:
            raise AttributeError()

    @property   # read-only
    def engine_ver(self):
        """str: Return PyRevitLoader.ScriptExecutor hardcoded version."""
        if PyRevitLoader:
            return PyRevitLoader.ScriptExecutor.EngineVersion

    @property  # read-only
    def first_load(self):
```

(continues on next page)

(continued from previous page)

```
        """bool: Check whether pyrevit is not running in pyrevit command."""
        # if no output window is set by the executor, it means that pyRevit
        # is loading at Revit startup (not reloading)
        return True if EXEC_PARAMS.window_handle is None else False

    @property   # read-only
    def pyrevit_command(self):
        """``PyRevitBaseClasses.PyRevitCommandRuntime``: Return command."""
        try:
            return __externalcommand__
        except NameError:
            return None

    @property   # read-only
    def forced_debug_mode(self):
        """bool: Check if command is in debug mode."""
        if self.pyrevit_command:
            return self.pyrevit_command.DebugMode
        else:
            return False

    @property   # read-only
    def executed_from_ui(self):
        """bool: Check if command was executed from ui."""
        if self.pyrevit_command:
            return self.pyrevit_command.ExecutedFromUI
        else:
            return False

    @property   # read
    def window_handle(self):
        """``PyRevitBaseClasses.ScriptOutput``: Return output window."""
        if self.pyrevit_command:
            return self.pyrevit_command.OutputWindow

    @property   # read-only
    def command_name(self):
        """str: Return current command name."""
        if '__commandname__' in __builtins__ \
                and __builtins__['__commandname__']:
            return __builtins__['__commandname__']
        elif self.pyrevit_command:
            return self.pyrevit_command.CommandName

    @property   # read-only
    def command_path(self):
        """str: Return current command path."""
        if '__commandpath__' in __builtins__ \
                and __builtins__['__commandpath__']:
            return __builtins__['__commandpath__']
        elif self.pyrevit_command:
            return op.dirname(self.pyrevit_command.ScriptSourceFile)

    @property
    def command_data(self):
        """``ExternalCommandData``: Return current command data."""
        if self.pyrevit_command:
```

(continues on next page)

(continued from previous page)

```
            return self.pyrevit_command.CommandData

    @property
    def doc_mode(self):
        """bool: Check if pyrevit is running by doc generator."""
        try:
            return __sphinx__
        except NameError:
            return False

    @property
    def command_mode(self):
        """bool: Check if pyrevit is running in pyrevit command context."""
        return self.pyrevit_command is not None

    @property
    def result_dict(self):
        """``Dictionary<String, String>``: Return results dict for logging."""
        if self.pyrevit_command:
            return self.pyrevit_command.GetResultsDictionary()


# create an instance of _ExecutorParams wrapping current runtime.
EXEC_PARAMS = _ExecutorParams()


# ------------------------------------------------------------------------------
# config user environment paths
# ------------------------------------------------------------------------------
# user env paths
USER_ROAMING_DIR = os.getenv('appdata')
USER_SYS_TEMP = os.getenv('temp')
USER_DESKTOP = op.expandvars('%userprofile%\\desktop')

# create paths for pyrevit files
if EXEC_PARAMS.doc_mode:
    PYREVIT_APP_DIR = PYREVIT_VERSION_APP_DIR = ' '
else:
    # pyrevit file directory
    PYREVIT_APP_DIR = op.join(USER_ROAMING_DIR, PYREVIT_ADDON_NAME)
    PYREVIT_VERSION_APP_DIR = op.join(PYREVIT_APP_DIR, HOST_APP.version)

    # add runtime paths to sys.paths
    # this will allow importing any dynamically compiled DLLs that
    # would be placed under this paths.
    for pyrvt_app_dir in [PYREVIT_APP_DIR, PYREVIT_VERSION_APP_DIR]:
        if not op.isdir(pyrvt_app_dir):
            try:
                os.mkdir(pyrvt_app_dir)
                sys.path.append(pyrvt_app_dir)
            except Exception as err:
                raise PyRevitException('Can not access pyRevit '
                                       'folder at: {} | {}'
                                       .format(pyrvt_app_dir, err))
        else:
            sys.path.append(pyrvt_app_dir)
```

(continues on next page)

(continued from previous page)

```
# ------------------------------------------------------------------------------
# standard prefixes for naming pyrevit files (config, appdata and temp files)
# ------------------------------------------------------------------------------
if EXEC_PARAMS.doc_mode:
    PYREVIT_FILE_PREFIX_UNIVERSAL = PYREVIT_FILE_PREFIX = \
        PYREVIT_FILE_PREFIX_STAMPED = None
    PYREVIT_FILE_PREFIX_UNIVERSAL_USER = PYREVIT_FILE_PREFIX_USER = \
        PYREVIT_FILE_PREFIX_STAMPED_USER = None
else:
    # e.g. pyRevit_
    PYREVIT_FILE_PREFIX_UNIVERSAL = '{}'.format(PYREVIT_ADDON_NAME)

    # e.g. pyRevit_2018_
    PYREVIT_FILE_PREFIX = '{}_{}'.format(PYREVIT_ADDON_NAME,
                                         HOST_APP.version)

    # e.g. pyRevit_2018_14422_
    PYREVIT_FILE_PREFIX_STAMPED = '{}_{}_{}'.format(PYREVIT_ADDON_NAME,
                                                    HOST_APP.version,
                                                    HOST_APP.proc_id)

    # e.g. pyRevit_eirannejad_
    PYREVIT_FILE_PREFIX_UNIVERSAL_USER = '{}_{}'.format(PYREVIT_ADDON_NAME,
                                                        HOST_APP.username)

    # e.g. pyRevit_2018_eirannejad_
    PYREVIT_FILE_PREFIX_USER = '{}_{}_{}'.format(PYREVIT_ADDON_NAME,
                                                 HOST_APP.version,
                                                 HOST_APP.username)

    # e.g. pyRevit_2018_eirannejad_14422_
    PYREVIT_FILE_PREFIX_STAMPED_USER = '{}_{}_{}_{}'.format(PYREVIT_ADDON_NAME,
                                                            HOST_APP.version,
                                                            HOST_APP.username,
                                                            HOST_APP.proc_id)
```

CHAPTER 11

pyrevit.api

11.1 Usage

```
from pyrevit.api import AdWindows
from pyrevit.api import NSJson
```

11.2 Documentation

Provide access to Revit API.

11.3 Implementation

```
"""Provide access to Revit API."""

from pyrevit.framework import clr

clr.AddReference('RevitAPI')
clr.AddReference('RevitAPIUI')
clr.AddReference('AdWindows')
clr.AddReference('UIFramework')
clr.AddReference('UIFrameworkServices')
clr.AddReference('Newtonsoft.Json')

# pylama:ignore=E402,W0611
# pylama ignore imports not on top and not used
import Autodesk.Internal as AdInternal
import Autodesk.Private as AdPrivate
import Autodesk.Windows as AdWindows
```

(continues on next page)

(continued from previous page)

```
import UIFramework
import UIFrameworkServices

import Newtonsoft.Json as NSJson

import Autodesk.Revit.Attributes as Attributes

import Autodesk.Revit.DB as DB
import Autodesk.Revit.UI as UI
```

CHAPTER 12

pyrevit.framework

12.1 Usage

```
from pyrevit.framework import Assembly, Windows
```

12.2 Documentation

Provide access to DotNet Framework.

pyrevit.framework.**get_type**(*fw_object*)
Return CLR type of an object.

Parameters **fw_object** – Dotnet Framework Object Instance

12.3 Implementation

```
"""Provide access to DotNet Framework."""

import clr

import System


clr.AddReference("System.Core")
clr.AddReference('System.Management')
clr.AddReferenceByPartialName('System.Windows.Forms')
clr.AddReferenceByPartialName('System.Drawing')
clr.AddReference('PresentationCore')
clr.AddReference('PresentationFramework')
```

(continues on next page)

(continued from previous page)

```
clr.AddReference('System.Xml.Linq')
clr.AddReferenceByPartialName('WindowsBase')

# add linq extensions?
clr.ImportExtensions(System.Linq)

# pylama:ignore=E402,W0611
# pylama ignore imports not on top and not used
from System import AppDomain, Version
from System import Type
from System import Uri, Guid
from System import EventHandler
from System import Array, IntPtr
from System.Collections import IEnumerator, IEnumerable
from System.Collections.Generic import List, Dictionary
from System import DateTime, DateTimeOffset

from System import Diagnostics
from System.Diagnostics import Process

from System import Reflection
from System.Reflection import Assembly, AssemblyName
from System.Reflection import TypeAttributes, MethodAttributes
from System.Reflection import CallingConventions
from System.Reflection import BindingFlags
from System.Reflection.Emit import AssemblyBuilderAccess
from System.Reflection.Emit import CustomAttributeBuilder, OpCodes

from System import IO
from System.IO import IOException, DriveInfo, Path, StringReader

from System import Net
from System.Net import WebClient, WebRequest, WebProxy

from System import Drawing
from System import Windows
from System.Windows import Forms
from System.Windows.Forms import Clipboard
from System.Windows import Controls
from System.Windows import Media
from System.Windows import Threading
from System.Windows import Interop
from System.Windows.Media import Imaging, SolidColorBrush, Color

from System import Math

from System.CodeDom import Compiler
from Microsoft.CSharp import CSharpCodeProvider

from System.Management import ManagementObjectSearcher

from System.Runtime.Serialization import FormatterServices

clr.AddReference('IronPython.Wpf')
import wpf
```

(continues on next page)

(continued from previous page)

```
def get_type(fw_object):
    """Return CLR type of an object.

    Args:
        fw_object: Dotnet Framework Object Instance
    """
    return clr.GetClrType(fw_object)
```

CHAPTER 13

pyrevit.script

13.1 Usage

See *pyrevit.script Module*

```
from pyrevit import script

script.clipboard_copy('some text')
data = script.journal_read('data-key')
script.exit()
```

13.2 Documentation

Provide basic utilities for pyRevit scripts.

pyrevit.script.**clipboard_copy**(*string_to_copy*)
Copy string to Windows Clipboard.

pyrevit.script.**exit**()
Stop the script execution and exit.

pyrevit.script.**get_bundle_file**(*file_name*)
Return full path to file under current script bundle.

Parameters **file_name** (*str*) – bundle file name

Returns full bundle file path

Return type str

pyrevit.script.**get_button**()
Find and return current script ui button.

Returns ui button object

Return type pyrevit.coreutils.ribbon._PyRevitRibbonButton

pyrevit.script.**get_config**()
Create and return config section parser object for current script.

Returns Config section parser object

Return type pyrevit.coreutils.configparser.PyRevitConfigSectionParser

pyrevit.script.**get_data_file**(*file_id*, *file_ext*, *add_cmd_name=False*)
Return filename to be used by a user script to store data.

File name is generated in this format: pyRevit_{Revit Version}_{file_id}.{file_ext}

Example

```
>>> script.get_document_data_file('mydata', 'data')
... '.../pyRevit_2018_mydata.data'
>>> script.get_document_data_file('mydata', 'data', add_cmd_name=True)
... '.../pyRevit_2018_Command Name_mydata.data'
```

Data files are not cleaned up at pyRevit startup. Script should manage cleaning up these files.

Parameters

- **file_id** (*str*) – unique id for the filename
- **file_ext** (*str*) – file extension
- **add_cmd_name** (*bool, optional*) – add command name to file name

Returns full file path

Return type str

pyrevit.script.**get_document_data_file**(*file_id*, *file_ext*, *add_cmd_name=False*)
Return filename to be used by a user script to store data.

File name is generated in this format: pyRevit_{Revit Version}_{file_id}_{Project Name}.{file_ext}

Example

```
>>> script.get_document_data_file('mydata', 'data')
... '.../pyRevit_2018_mydata_Project1.data'
>>> script.get_document_data_file('mydata', 'data', add_cmd_name=True)
... '.../pyRevit_2018_Command Name_mydata_Project1.data'
```

Document data files are not cleaned up at pyRevit startup. Script should manage cleaning up these files.

Parameters

- **file_id** (*str*) – unique id for the filename
- **file_ext** (*str*) – file extension
- **add_cmd_name** (*bool, optional*) – add command name to file name

Returns full file path

Return type str

pyrevit.script.**get_info**()
Return info on current pyRevit command.

Returns Command info object

Return type pyrevit.extensions.genericcomps.GenericUICommand

pyrevit.script.**get_instance_data_file**(*file_id*, *add_cmd_name=False*)
Return filename to be used by a user script to store data.

File name is generated in this format: pyRevit_{Revit Version}_{Process Id}_{file_id}.{file_ext}

Example

```
>>> script.get_document_data_file('mydata')
... '.../pyRevit_2018_6684_mydata.tmp'
>>> script.get_document_data_file('mydata', add_cmd_name=True)
... '.../pyRevit_2018_6684_Command Name_mydata.tmp'
```

Instance data files are cleaned up at pyRevit startup.

Parameters

- **file_id** (*str*) – unique id for the filename
- **add_cmd_name** (*bool, optional*) – add command name to file name

Returns full file path

Return type str

pyrevit.script.**get_logger**()
Create and return logger named for current script.

Returns Logger object

Return type pyrevit.coreutils.logger.LoggerWrapper

pyrevit.script.**get_output**()
Return object wrapping output window for current script.

Returns Output wrapper object

Return type *pyrevit.output.PyRevitOutputWindow*

pyrevit.script.**get_pyrevit_version**()
Return pyRevit version.

Returns pyRevit version provider

Return type pyrevit.versionmgr.PyRevitVersion

pyrevit.script.**get_results**()
Return command results dictionary for logging.

Returns Command results dict

Return type pyrevit.usagelog.record.CommandCustomResults

pyrevit.script.**get_universal_data_file**(*file_id*, *file_ext*, *add_cmd_name=False*)
Return filename to be used by a user script to store data.

File name is generated in this format: pyRevit_{file_id}.{file_ext}

Example

```
>>> script.get_document_data_file('mydata', 'data')
... '.../pyRevit_mydata.data'
>>> script.get_document_data_file('mydata', 'data', add_cmd_name=True)
... '.../pyRevit_Command Name_mydata.data'
```

Universal data files are not cleaned up at pyRevit startup. Script should manage cleaning up these files.

Parameters

- **file_id** (*str*) – unique id for the filename
- **file_ext** (*str*) – file extension
- **add_cmd_name** (*bool, optional*) – add command name to file name

Returns full file path

Return type str

pyrevit.script.**journal_read**(*data_key*)
Read value for provided key from active Revit journal.

Parameters **data_key** (*str*) – data key

Returns data value string

Return type str

pyrevit.script.**journal_write**(*data_key*, *msg*)
Write key and value to active Revit journal for current command.

Parameters

- **data_key** (*str*) – data key
- **msg** (*str*) – data value string

pyrevit.script.**load_index**(*index_file='index.html'*)
Load html file into output window.

This method expects index.html file in the current command bundle, unless full path to an html file is provided.

Parameters **index_file** (*str, optional*) – full path of html file.

pyrevit.script.**open_url**(*url*)
Open url in a new tab in default webbrowser.

pyrevit.script.**save_config**()
Save pyRevit config.

Scripts should call this to save any changes they have done to their config section object received from script.get_config() method.

pyrevit.script.**show_file_in_explorer**(*file_path*)
Show file in Windows Explorer.

pyrevit.script.**toggle_icon**(*new_state*, *on_icon_path=None*, *off_icon_path=None*)
Set the state of button icon (on or off).

This method expects on.png and off.png in command bundle for on and off icon states, unless full path of icon states are provided.

Parameters

- **new_state** (*bool*) – state of the ui button icon.
- **on_icon_path** (*str, optional*) – full path of icon for on state. default='on.png'
- **off_icon_path** (*str, optional*) – full path of icon for off state. default='off.png'

13.3 Implementation

```
"""Provide basic utilities for pyRevit scripts."""

import sys
import os
import os.path as op

from pyrevit import EXEC_PARAMS
from pyrevit.coreutils import logger
from pyrevit.coreutils import appdata
from pyrevit import framework
from pyrevit import revit
from pyrevit import output

# suppress any warning generated by native or third-party modules
import warnings
warnings.filterwarnings("ignore")


mlogger = logger.get_logger(__name__)


def get_info():
    """Return info on current pyRevit command.

    Returns:
        :obj:`pyrevit.extensions.genericcomps.GenericUICommand`:
            Command info object
    """
    from pyrevit.extensions.extensionmgr import get_command_from_path
    return get_command_from_path(EXEC_PARAMS.command_path)


def get_results():
    """Return command results dictionary for logging.

    Returns:
        :obj:`pyrevit.usagelog.record.CommandCustomResults`:
            Command results dict
    """
    from pyrevit.usagelog.record import CommandCustomResults
    return CommandCustomResults()


def get_pyrevit_version():
    """Return pyRevit version.

    Returns:
        :obj:`pyrevit.versionmgr.PyRevitVersion`: pyRevit version provider
```

(continues on next page)

(continued from previous page)

```
    """
    from pyrevit.versionmgr import PYREVIT_VERSION
    return PYREVIT_VERSION


def get_logger():
    """Create and return logger named for current script.

    Returns:
        :obj:`pyrevit.coreutils.logger.LoggerWrapper`: Logger object
    """
    return logger.get_logger(EXEC_PARAMS.command_name)


def get_output():
    """Return object wrapping output window for current script.

    Returns:
        :obj:`pyrevit.output.PyRevitOutputWindow`: Output wrapper object
    """
    return output.get_output()


def get_config():
    """Create and return config section parser object for current script.

    Returns:
        :obj:`pyrevit.coreutils.configparser.PyRevitConfigSectionParser`:
            Config section parser object
    """
    from pyrevit.userconfig import user_config
    script_cfg_postfix = 'config'

    try:
        return user_config.get_section(EXEC_PARAMS.command_name +
                                       script_cfg_postfix)
    except Exception:
        return user_config.add_section(EXEC_PARAMS.command_name +
                                       script_cfg_postfix)


def save_config():
    """Save pyRevit config.

    Scripts should call this to save any changes they have done to their
    config section object received from script.get_config() method.
    """
    from pyrevit.userconfig import user_config
    user_config.save_changes()


def get_universal_data_file(file_id, file_ext, add_cmd_name=False):
    """Return filename to be used by a user script to store data.

    File name is generated in this format:
    ``pyRevit_{file_id}.{file_ext}``
```

(continues on next page)

(continued from previous page)

```
    Example:
        >>> script.get_document_data_file('mydata', 'data')
        ... '.../pyRevit_mydata.data'
        >>> script.get_document_data_file('mydata', 'data', add_cmd_name=True)
        ... '.../pyRevit_Command Name_mydata.data'

    Universal data files are not cleaned up at pyRevit startup.
    Script should manage cleaning up these files.

    Args:
        file_id (str): unique id for the filename
        file_ext (str): file extension
        add_cmd_name (bool, optional): add command name to file name

    Returns:
        str: full file path
    """
    if add_cmd_name:
        script_file_id = '{}_{}'.format(EXEC_PARAMS.command_name, file_id)
    else:
        script_file_id = file_id

    return appdata.get_universal_data_file(script_file_id, file_ext)


def get_data_file(file_id, file_ext, add_cmd_name=False):
    """Return filename to be used by a user script to store data.

    File name is generated in this format:
    ``pyRevit_{Revit Version}_{file_id}.{file_ext}``

    Example:
        >>> script.get_document_data_file('mydata', 'data')
        ... '.../pyRevit_2018_mydata.data'
        >>> script.get_document_data_file('mydata', 'data', add_cmd_name=True)
        ... '.../pyRevit_2018_Command Name_mydata.data'

    Data files are not cleaned up at pyRevit startup.
    Script should manage cleaning up these files.

    Args:
        file_id (str): unique id for the filename
        file_ext (str): file extension
        add_cmd_name (bool, optional): add command name to file name

    Returns:
        str: full file path
    """
    if add_cmd_name:
        script_file_id = '{}_{}'.format(EXEC_PARAMS.command_name, file_id)
    else:
        script_file_id = file_id

    return appdata.get_data_file(script_file_id, file_ext)


def get_instance_data_file(file_id, add_cmd_name=False):
```

(continues on next page)

(continued from previous page)

```
    """Return filename to be used by a user script to store data.

    File name is generated in this format:
    ``pyRevit_{Revit Version}_{Process Id}_{file_id}.{file_ext}``

    Example:
        >>> script.get_document_data_file('mydata')
        ... '.../pyRevit_2018_6684_mydata.tmp'
        >>> script.get_document_data_file('mydata', add_cmd_name=True)
        ... '.../pyRevit_2018_6684_Command Name_mydata.tmp'

    Instance data files are cleaned up at pyRevit startup.

    Args:
        file_id (str): unique id for the filename
        add_cmd_name (bool, optional): add command name to file name

    Returns:
        str: full file path
    """
    if add_cmd_name:
        script_file_id = '{}_{}'.format(EXEC_PARAMS.command_name, file_id)
    else:
        script_file_id = file_id

    return appdata.get_instance_data_file(script_file_id)


def get_document_data_file(file_id, file_ext, add_cmd_name=False):
    """Return filename to be used by a user script to store data.

    File name is generated in this format:
    ``pyRevit_{Revit Version}_{file_id}_{Project Name}.{file_ext}``

    Example:
        >>> script.get_document_data_file('mydata', 'data')
        ... '.../pyRevit_2018_mydata_Project1.data'
        >>> script.get_document_data_file('mydata', 'data', add_cmd_name=True)
        ... '.../pyRevit_2018_Command Name_mydata_Project1.data'

    Document data files are not cleaned up at pyRevit startup.
    Script should manage cleaning up these files.

    Args:
        file_id (str): unique id for the filename
        file_ext (str): file extension
        add_cmd_name (bool, optional): add command name to file name

    Returns:
        str: full file path
    """
    proj_info = revit.get_project_info()

    if add_cmd_name:
        script_file_id = '{}_{}_{}'.format(EXEC_PARAMS.command_name,
                                           file_id,
                                           proj_info.filename
```

(continues on next page)

(continued from previous page)

```
                                            or proj_info.name)
    else:
        script_file_id = '{}_{}'.format(file_id,
                                        proj_info.filename
                                        or proj_info.name)

    return appdata.get_data_file(script_file_id, file_ext)


def get_bundle_file(file_name):
    """Return full path to file under current script bundle.

    Args:
        file_name (str): bundle file name

    Returns:
        str: full bundle file path
    """
    return op.join(EXEC_PARAMS.command_path, file_name)


def journal_write(data_key, msg):
    """Write key and value to active Revit journal for current command.

    Args:
        data_key (str): data key
        msg (str): data value string
    """
    # Get the StringStringMap class which can write data into.
    # noinspection PyUnresolvedReferences
    data_map = EXEC_PARAMS.command_data.JournalData
    data_map.Clear()

    # Begin to add the support data
    data_map.Add(data_key, msg)


def journal_read(data_key):
    """Read value for provided key from active Revit journal.

    Args:
        data_key (str): data key

    Returns:
        str: data value string
    """
    # Get the StringStringMap class which can write data into.
    # noinspection PyUnresolvedReferences
    data_map = EXEC_PARAMS.command_data.JournalData

    # Begin to get the support data
    return data_map[data_key]


def get_button():
    """Find and return current script ui button.
```

(continues on next page)

(continued from previous page)

```
    Returns:
        :obj:`pyrevit.coreutils.ribbon._PyRevitRibbonButton`: ui button object
    """
    from pyrevit.coreutils.ribbon import get_current_ui
    pyrvt_tabs = get_current_ui().get_pyrevit_tabs()
    for tab in pyrvt_tabs:
        button = tab.find_child(EXEC_PARAMS.command_name)
        if button:
            return button
    return None


def toggle_icon(new_state, on_icon_path=None, off_icon_path=None):
    """Set the state of button icon (on or off).

    This method expects on.png and off.png in command bundle for on and off
    icon states, unless full path of icon states are provided.

    Args:
        new_state (bool): state of the ui button icon.
        on_icon_path (str, optional): full path of icon for on state.
                                      default='on.png'
        off_icon_path (str, optional): full path of icon for off state.
                                       default='off.png'
    """
    # find the ui button
    uibutton = get_button()
    if not uibutton:
        mlogger.debug('Can not find ui button.')
        return

    # get icon for on state
    if not on_icon_path:
        on_icon_path = get_bundle_file('on.png')
        if not on_icon_path:
            mlogger.debug('Script does not have icon for on state.')
            return

    # get icon for off state
    if not off_icon_path:
        off_icon_path = get_bundle_file('off.png')
        if not off_icon_path:
            mlogger.debug('Script does not have icon for on state.')
            return

    icon_path = on_icon_path if new_state else off_icon_path
    mlogger.debug('Setting icon state to: {} ({})'
                  .format(new_state, icon_path))
    uibutton.set_icon(icon_path)


def exit():
    """Stop the script execution and exit."""
    sys.exit()


def show_file_in_explorer(file_path):
```

(continues on next page)

(continued from previous page)

```
    """Show file in Windows Explorer."""
    import subprocess
    subprocess.Popen(r'explorer /select,"{}"'
                     .format(os.path.normpath(file_path)))


def open_url(url):
    """Open url in a new tab in default webbrowser."""
    import webbrowser
    return webbrowser.open_new_tab(url)


def clipboard_copy(string_to_copy):
    """Copy string to Windows Clipboard."""
    framework.Clipboard.SetText(string_to_copy)


def load_index(index_file='index.html'):
    """Load html file into output window.

    This method expects index.html file in the current command bundle,
    unless full path to an html file is provided.

    Args:
        index_file (str, optional): full path of html file.
    """
    outputwindow = get_output()
    if not op.isfile(index_file):
        index_file = get_bundle_file(index_file)
    outputwindow.open_page(index_file)
```

CHAPTER 14

pyrevit.userconfig

14.1 Usage

This module handles the reading and writing of the pyRevit configuration files. It's been used extensively by pyRevit sub-modules. `user_config` is set up automatically in the global scope by this module and can be imported into scripts and other modules to access the default configurations.

Example:

```
>>> from pyrevit.userconfig import user_config
>>> user_config.add_section('newsection')
>>> user_config.newsection.property = value
>>> user_config.newsection.get('property', default_value)
>>> user_config.save_changes()
```

The `user_config` object is also the destination for reading and writing configuration by pyRevit scripts through `get_config()` of *pyrevit.script* module. Here is the function source:

```
def get_config():
    """Create and return config section parser object for current script.

    Returns:
        :obj:`pyrevit.coreutils.configparser.PyRevitConfigSectionParser`:
            Config section parser object
    """
    from pyrevit.userconfig import user_config
    script_cfg_postfix = 'config'

    try:
        return user_config.get_section(EXEC_PARAMS.command_name +
                                       script_cfg_postfix)
    except Exception:
        return user_config.add_section(EXEC_PARAMS.command_name +
                                       script_cfg_postfix)
```

Example:

```
>>> from pyrevit import script
>>> cfg = script.get_config()
>>> cfg.property = value
>>> cfg.get('property', default_value)
>>> script.save_config()
```

14.2 Documentation

Handle reading and parsing, writin and saving of all user configurations.

All other modules use this module to query user config.

class pyrevit.userconfig.**PyRevitConfig**(*cfg_file_path=None*)

Provide read/write access to pyRevit configuration.

Parameters **cfg_file_path** (*str*) – full path to config file to be used.

Example

```
>>> cfg = PyRevitConfig(cfg_file_path)
>>> cfg.add_section('sectionname')
>>> cfg.sectionname.property = value
>>> cfg.sectionname.get('property', default_value)
>>> cfg.save_changes()
```

get_config_version()

Return version of config file used for change detection.

get_ext_root_dirs()

Return a list of external extension directories set by the user.

Returns list of strings. External user extension directories.

Return type list

save_changes()

Save user config into associated config file.

14.3 Implementation

```
"""Handle reading and parsing, writin and saving of all user configurations.

All other modules use this module to query user config.
"""

import os
import os.path as op
import shutil

from pyrevit import EXEC_PARAMS, EXTENSIONS_DEFAULT_DIR
from pyrevit.framework import IOException
```

(continues on next page)

(continued from previous page)

```
from pyrevit.coreutils import touch
import pyrevit.coreutils.appdata as appdata
from pyrevit.coreutils.configparser import PyRevitConfigParser
from pyrevit.coreutils.logger import get_logger, set_file_logging
from pyrevit.versionmgr.upgrade import upgrade_user_config


logger = get_logger(__name__)


INIT_SETTINGS_SECTION = 'core'


# location for default pyRevit config files
if not EXEC_PARAMS.doc_mode:
    ADMIN_CONFIG_DIR = op.join(os.getenv('programdata'), 'pyRevit')

    # setup config file name and path
    CONFIG_FILE_PATH = appdata.get_universal_data_file(file_id='config',
                                                       file_ext='ini')
    logger.debug('User config file: {}'.format(CONFIG_FILE_PATH))
else:
    ADMIN_CONFIG_DIR = CONFIG_FILE_PATH = None


# ==============================================================================
# fix obsolete config file naming
# config file (and all appdata files) used to include username in the filename
# this fixes the existing config file with obsolete naming, to new format
# pylama:ignore=E402
from pyrevit import PYREVIT_APP_DIR, PYREVIT_FILE_PREFIX_UNIVERSAL_USER

OBSOLETE_CONFIG_FILENAME = '{}_{}'.format(PYREVIT_FILE_PREFIX_UNIVERSAL_USER,
                                          'config.ini')
OBSOLETE_CONFIG_FILEPATH = op.join(PYREVIT_APP_DIR, OBSOLETE_CONFIG_FILENAME)

if op.exists(OBSOLETE_CONFIG_FILEPATH):
    try:
        os.rename(OBSOLETE_CONFIG_FILEPATH, CONFIG_FILE_PATH)
    except Exception as rename_err:
        logger.error('Failed to update the config file name to new format. '
                     'A new configuration file has been created for you '
                     'under \n{}'
                     '\nYour previous pyRevit configuration file still '
                     'existing under the same folder. Please close Revit, '
                     'open both configuration files and copy and paste '
                     'settings from the old config file to new config file. '
                     'Then you can remove the old config file as pyRevit '
                     'will not be using that anymore. | {}'
                     .format(CONFIG_FILE_PATH, rename_err))
# end fix obsolete config file naming
# ==============================================================================


class PyRevitConfig(PyRevitConfigParser):
    """Provide read/write access to pyRevit configuration.
```

(continues on next page)

(continued from previous page)

```
    Args:
        cfg_file_path (str): full path to config file to be used.

    Example:
        >>> cfg = PyRevitConfig(cfg_file_path)
        >>> cfg.add_section('sectionname')
        >>> cfg.sectionname.property = value
        >>> cfg.sectionname.get('property', default_value)
        >>> cfg.save_changes()
    """

    def __init__(self, cfg_file_path=None):
        """Load settings from provided config file and setup parser."""
        self.config_file = cfg_file_path

        # try opening and reading config file in order.
        PyRevitConfigParser.__init__(self, cfg_file_path=cfg_file_path)

        # set log mode on the logger module based on
        # user settings (overriding the defaults)
        self._update_env()

    def _update_env(self):
        # update the debug level based on user config
        logger.reset_level()

        try:
            # first check to see if command is not in forced debug mode
            if not EXEC_PARAMS.forced_debug_mode:
                if self.core.debug:
                    logger.set_debug_mode()
                    logger.debug('Debug mode is enabled in user settings.')
                elif self.core.verbose:
                    logger.set_verbose_mode()

            set_file_logging(self.core.filelogging)
        except Exception as env_update_err:
            logger.debug('Error updating env variable per user config. | {}'
                         .format(env_update_err))

    def get_config_version(self):
        """Return version of config file used for change detection."""
        return self.get_config_file_hash()

    def get_ext_root_dirs(self):
        """Return a list of external extension directories set by the user.

        Returns:
            :obj:`list`: list of strings. External user extension directories.
        """
        dir_list = list()
        dir_list.append(EXTENSIONS_DEFAULT_DIR)
        try:
            dir_list.extend([p for p in self.core.userextensions])
        except Exception as read_err:
            logger.error('Error reading list of user extension folders. | {}'
```

(continues on next page)

(continued from previous page)

```
                        .format(read_err))

        return dir_list

    def save_changes(self):
        """Save user config into associated config file."""
        try:
            PyRevitConfigParser.save(self, self.config_file)
        except Exception as save_err:
            logger.error('Can not save user config to: {} | {}'
                         .format(self.config_file, save_err))

        # adjust environment per user configurations
        self._update_env()


def _set_hardcoded_config_values(parser):
    """Set default config values for user configuration.

    Args:
        parser (:obj:`pyrevit.userconfig.PyRevitConfig`):
            parser to accept the default values
    """
    # hard-coded values
    parser.add_section('core')
    parser.core.checkupdates = False
    parser.core.verbose = True
    parser.core.debug = False
    parser.core.filelogging = True
    parser.core.startuplogtimeout = 10
    parser.core.userextensions = []
    parser.core.compilecsharp = True
    parser.core.compilevb = True
    parser.core.loadbeta = False
    parser.core.rocketmode = False


def _get_default_config_parser(config_file_path):
    """Create a user settings file.

    Args:
        config_file_path (str): config file full name and path

    Returns:
        :obj:`pyrevit.userconfig.PyRevitConfig`: pyRevit config file handler
    """
    logger.debug('Creating default config file at: {} '
                 .format(CONFIG_FILE_PATH))
    touch(config_file_path)

    try:
        parser = PyRevitConfig(cfg_file_path=config_file_path)
    except Exception as read_err:
        # can not create default user config file under appdata folder
        logger.debug('Can not create config file under: {} | {}'
                     .format(config_file_path, read_err))
        parser = PyRevitConfig()
```

(continues on next page)

(continued from previous page)

```
    # set hard-coded values
    _set_hardcoded_config_values(parser)

    # save config into config file
    parser.save_changes()
    logger.debug('Default config saved to: {}'
                 .format(config_file_path))

    return parser


def _setup_admin_config():
    """Setup the default config file with hardcoded values."""
    if not op.exists(CONFIG_FILE_PATH) \
            and op.isdir(ADMIN_CONFIG_DIR):
        for entry in os.listdir(ADMIN_CONFIG_DIR):
            if entry.endswith('.ini'):
                sourcecfg = op.join(ADMIN_CONFIG_DIR, entry)
                try:
                    shutil.copyfile(sourcecfg, CONFIG_FILE_PATH)
                    logger.debug('Configured from admin file: {}'
                                 .format(sourcecfg))
                except Exception as copy_err:
                    logger.debug('Error copying admin config file: {}'
                                 .format(sourcecfg))
                return True


if not EXEC_PARAMS.doc_mode:
    # check to see if there is any config file provided by admin
    # if yes, copy that and use as default
    _setup_admin_config()

    # read user config, or setup default config file if not available
    # this pushes reading settings at first import of this module.
    try:
        user_config = PyRevitConfig(cfg_file_path=CONFIG_FILE_PATH)
        upgrade_user_config(user_config)
    except Exception as cfg_err:
        logger.debug('Can not read existing confing file at: {} | {}'
                     .format(CONFIG_FILE_PATH, cfg_err))
        user_config = _get_default_config_parser(CONFIG_FILE_PATH)
else:
    user_config = None
```

CHAPTER 15

pyrevit.output

Provides access and control over the pyRevit output window.

15.1 pyrevit.output

15.1.1 Usage

This module provides access to the output window for the currently running pyRevit command. The proper way to access this wrapper object is through the `get_output()` of *pyrevit.script* module. This method, in return uses the *pyrevit.output* module to get access to the output wrapper.

Example:

```
>>> from pyrevit import script
>>> output = script.get_output()
```

Here is the source of *pyrevit.script.get_output()*. As you can see this functions calls the *pyrevit.output.get_output()* to receive the output wrapper.

```
def get_output():
    """Return object wrapping output window for current script.

    Returns:
        :obj:`pyrevit.output.PyRevitOutputWindow`: Output wrapper object
    """
    return output.get_output()
```

15.1.2 Documentation

Provide access to output window and its functionality.

class pyrevit.output.**PyRevitOutputWindow**

Wrapper to interact with the output window.

add_style(*style_code*, *attribs=None*)

Inject style tag into current html head of the output window.

Parameters

- **style_code** (*str*) – css styling code
- **attribs** (dict) – dictionary of attribute names and value

Example

```
>>> output = pyrevit.output.get_output()
>>> output.add_style('body { color: blue; }')
```

close()

Close the window.

close_others(*all_open_outputs=False*)

Close all other windows that belong to the current command.

Parameters **all_open_outputs** (*bool*) – Close all any other windows if True

static emojize(*md_str*)

Replace emoji codes with emoji images and print.

Parameters **md_str** (*str*) – string containing emoji code

Example

```
>>> output = pyrevit.output.get_output()
>>> output.emojize('Process completed. :thumbs_up:')
```

get_head_html()

str: Return inner code of html head element.

get_height()

int: Return current window height.

get_title()

str: Return current window title.

get_width()

int: Return current window width.

hide()

Hide the window.

inject_script(*script_code*, *attribs=None*)

Inject script tag into current html head of the output window.

Parameters

- **script_code** (*str*) – javascript code
- **attribs** (dict) – dictionary of attribute names and value

Example

```
>>> output = pyrevit.output.get_output()
>>> output.inject_script('',   # no script since it's a link
                         {'src': js_script_file_path})
```

inject_to_head(*element_tag*, *element_contents*, *attribs=None*)
Inject html element to current html head of the output window.

Parameters

- **element_tag** (*str*) – html tag of the element e.g. 'div'
- **element_contents** (*str*) – html code of the element contents
- **attribs** (dict) – dictionary of attribute names and value

Example

```
>>> output = pyrevit.output.get_output()
>>> output.inject_to_head('script',
                          '',   # no script since it's a link
                          {'src': js_script_file_path})
```

insert_divider()
Add horizontal rule to the output window.

static linkify(*element_ids*, *title=None*)
Create clickable link for the provided element ids.

This method, creates the link but does not print it directly.

Parameters

- **element_ids** (list of *ElementId*) –
- **element_ids** – single or multiple ids
- **title** (*str*) – tile of the link. defaults to list of element ids

Example

```
>>> output = pyrevit.output.get_output()
>>> for idx, elid in enumerate(element_ids):
>>>     print('{}: {}'.format(idx+1, output.linkify(elid)))
```

lock_size()
Lock window size.

make_bar_chart()
PyRevitOutputChart: Return bar chart object.

make_bubble_chart()
PyRevitOutputChart: Return bubble chart object.

make_chart()
PyRevitOutputChart: Return chart object.

make_doughnut_chart()
PyRevitOutputChart: Return dougnut chart object.

make_line_chart()
PyRevitOutputChart: Return line chart object.

make_pie_chart()
PyRevitOutputChart: Return pie chart object.

make_polar_chart()
PyRevitOutputChart: Return polar chart object.

make_radar_chart()
PyRevitOutputChart: Return radar chart object.

make_stacked_chart()
PyRevitOutputChart: Return stacked chart object.

next_page()
Add hidden next page tag to the output window.

This is helpful to silently separate the output to multiple pages for better printing.

open_page(*dest_file*)
Open html page in output window.

Parameters **dest_file** (*str*) – full path of the target html file

open_url(*dest_url*)
Open url page in output window.

Parameters **dest_url** (*str*) – web url of the target page

output_id
str – Return id of the output window.

In current implementation, Id of output window is equal to the unique id of the pyRevit command it belongs to. This means that all output windows belonging to the same pyRevit command, will have identical output_id values.

output_uniqueid
str – Return unique id of the output window.

In current implementation, unique id of output window is a GUID string generated when the output window is opened. This id is unique to the instance of output window.

static print_code(*code_str*)
Print code to the output window with special formatting.

Example

```
>>> output = pyrevit.output.get_output()
>>> output.print_code('value = 12')
```

static print_html(*html_str*)
Add the html code to the output window.

Example

```
>>> output = pyrevit.output.get_output()
>>> output.print_html('<strong>Title</strong>')
```

static print_md(*md_str*)
Process markdown code and print to output window.

Example

```
>>> output = pyrevit.output.get_output()
>>> output.print_md('### Title')
```

print_table(*table_data*, *columns=[]*, *formats=[]*, *title=''*, *last_line_style=''*)
Print provided data in a table in output window.

Parameters

- **table_data** (list of iterables) – 2D array of data
- **title** (*str*) – table title
- **columns** (list str) – list of column names
- **formats** (list str) – column data formats
- **last_line_style** (*str*) – css style of last row

Example

```
>>> data = [
... ['row1', 'data', 'data', 80 ],
... ['row2', 'data', 'data', 45 ],
... ]
>>> output.print_table(
... table_data=data,
... title="Example Table",
... columns=["Row Name", "Column 1", "Column 2", "Percentage"],
... formats=['', '', '', '{}%'],
... last_line_style='color:red;'
... )
```

renderer
Return html renderer inside output window.

Returns System.Windows.Forms.WebBrowser (In current implementation)

reset_progress()
Reset output window progress bar to zero.

resize(*width*, *height*)
Resize window to the new width and height.

save_contents(*dest_file*)
Save html code of the window.

Parameters **dest_file** (*str*) – full path of the destination html file

self_destruct(*seconds*)
Set self-destruct (close window) timer.

Parameters **seconds** (*int*) – number of seconds after which window is closed.

set_font(*font_family*, *font_size*)
Set window font family to the new font family and size.

Parameters

- **font_family** (*str*) – font family name e.g. 'Courier New'
- **font_size** (*int*) – font size e.g. 16

set_height(*height*)
Set window height to the new height.

set_title(*new_title*)
Set window title to the new title.

set_width(*width*)
Set window width to the new width.

show()
Show the window.

update_progress(*cur_value*, *max_value*)
Activate and update the output window progress bar.

Parameters

- **cur_value** (*float*) – current progress value e.g. 50
- **max_value** (*float*) – total value e.g. 100

Example

```
>>> output = pyrevit.output.get_output()
>>> for i in range(100):
>>>     output.update_progress(i, 100)
```

window
PyRevitBaseClasses.ScriptOutput – Return output window object.

pyrevit.output.**get_default_stylesheet**()
Return default css stylesheet used by output window.

pyrevit.output.**get_output**()
pyrevit.output.PyRevitOutputWindow : Return output window.

pyrevit.output.**get_stylesheet**()
Return active css stylesheet used by output window.

pyrevit.output.**reset_stylesheet**()
Reset active stylesheet to default.

pyrevit.output.**set_stylesheet**(*stylesheet*)
Set active css stylesheet used by output window.

Parameters **stylesheet** (*str*) – full path to stylesheet file

15.1.3 Implementation

```
"""Provide access to output window and its functionality."""

from __future__ import print_function
import os.path as op
import itertools

from pyrevit import EXEC_PARAMS
from pyrevit import framework
from pyrevit import coreutils
from pyrevit.coreutils import logger
from pyrevit.coreutils import markdown, charts
from pyrevit.coreutils import emoji
from pyrevit.coreutils import envvars
from pyrevit.coreutils.loadertypes import EnvDictionaryKeys
from pyrevit.coreutils.loadertypes import ScriptOutputManager
from pyrevit.output import linkmaker
from pyrevit.userconfig import user_config


mlogger = logger.get_logger(__name__)


DEFAULT_STYLESHEET_NAME = 'outputstyles.css'


def set_stylesheet(stylesheet):
    """Set active css stylesheet used by output window.

    Args:
        stylesheet (str): full path to stylesheet file
    """
    envvars.set_pyrevit_env_var(EnvDictionaryKeys.outputStyleSheet,
                                stylesheet)


def get_stylesheet():
    """Return active css stylesheet used by output window."""
    return envvars.get_pyrevit_env_var(EnvDictionaryKeys.outputStyleSheet)


def get_default_stylesheet():
    """Return default css stylesheet used by output window."""
    return op.join(op.dirname(__file__), DEFAULT_STYLESHEET_NAME)


def reset_stylesheet():
    """Reset active stylesheet to default."""
    envvars.set_pyrevit_env_var(EnvDictionaryKeys.outputStyleSheet,
                                get_default_stylesheet())


# setup output window stylesheet
if not EXEC_PARAMS.doc_mode:
    active_stylesheet = \
        user_config.core.get_option('outputstylesheet',
```

(continues on next page)

(continued from previous page)

```
                                        default_value=get_default_stylesheet())
    set_stylesheet(active_stylesheet)


class PyRevitOutputWindow(object):
    """Wrapper to interact with the output window."""

    @property
    def window(self):
        """``PyRevitBaseClasses.ScriptOutput``: Return output window object."""
        return EXEC_PARAMS.window_handle

    @property
    def renderer(self):
        """Return html renderer inside output window.

        Returns:
            ``System.Windows.Forms.WebBrowser`` (In current implementation)
        """
        if self.window:
            return self.window.renderer

    @property
    def output_id(self):
        """str: Return id of the output window.

        In current implementation, Id of output window is equal to the
        unique id of the pyRevit command it belongs to. This means that all
        output windows belonging to the same pyRevit command, will have
        identical output_id values.
        """
        if self.window:
            return self.window.OutputId

    @property
    def output_uniqueid(self):
        """str: Return unique id of the output window.

        In current implementation, unique id of output window is a GUID string
        generated when the output window is opened. This id is unique to the
        instance of output window.
        """
        if self.window:
            return self.window.OutputUniqueId

    def _get_head_element(self):
        return self.renderer.Document.GetElementsByTagName('head')[0]

    def self_destruct(self, seconds):
        """Set self-destruct (close window) timer.

        Args:
            seconds (int): number of seconds after which window is closed.
        """
        if self.window:
            self.window.SelfDestructTimer(seconds)
```

(continues on next page)

(continued from previous page)

```
    def inject_to_head(self, element_tag, element_contents, attribs=None):
        """Inject html element to current html head of the output window.

        Args:
            element_tag (str): html tag of the element e.g. 'div'
            element_contents (str): html code of the element contents
            attribs (:obj:`dict`): dictionary of attribute names and value

        Example:
            >>> output = pyrevit.output.get_output()
            >>> output.inject_to_head('script',
                                      '',   # no script since it's a link
                                      {'src': js_script_file_path})
        """
        html_element = self.renderer.Document.CreateElement(element_tag)
        if element_contents:
            html_element.InnerHtml = element_contents

        if attribs:
            for attribute, value in attribs.items():
                html_element.SetAttribute(attribute, value)

        # inject the script into head
        head_el = self._get_head_element()
        head_el.AppendChild(html_element)

    def inject_script(self, script_code, attribs=None):
        """Inject script tag into current html head of the output window.

        Args:
            script_code (str): javascript code
            attribs (:obj:`dict`): dictionary of attribute names and value

        Example:
            >>> output = pyrevit.output.get_output()
            >>> output.inject_script('',   # no script since it's a link
                                     {'src': js_script_file_path})
        """
        self.inject_to_head('script', script_code, attribs=attribs)

    def add_style(self, style_code, attribs=None):
        """Inject style tag into current html head of the output window.

        Args:
            style_code (str): css styling code
            attribs (:obj:`dict`): dictionary of attribute names and value

        Example:
            >>> output = pyrevit.output.get_output()
            >>> output.add_style('body { color: blue; }')
        """
        self.inject_to_head('style', style_code, attribs=attribs)

    def get_head_html(self):
        """str: Return inner code of html head element."""
        return self._get_head_element().InnerHtml
```

(continues on next page)

(continued from previous page)

```
    def set_title(self, new_title):
        """Set window title to the new title."""
        if self.window:
            self.window.Text = new_title

    def set_width(self, width):
        """Set window width to the new width."""
        if self.window:
            self.window.Width = width

    def set_height(self, height):
        """Set window height to the new height."""
        if self.window:
            self.window.Height = height

    def set_font(self, font_family, font_size):
        """Set window font family to the new font family and size.

        Args:
            font_family (str): font family name e.g. 'Courier New'
            font_size (int): font size e.g. 16
        """
        # noinspection PyUnresolvedReferences
        self.renderer.Font = \
            framework.Drawing.Font(font_family,
                                   font_size,
                                   framework.Drawing.FontStyle.Regular,
                                   framework.Drawing.GraphicsUnit.Point)

    def resize(self, width, height):
        """Resize window to the new width and height."""
        self.set_width(width)
        self.set_height(height)

    def get_title(self):
        """str: Return current window title."""
        if self.window:
            return self.window.Text

    def get_width(self):
        """int: Return current window width."""
        if self.window:
            return self.window.Width

    def get_height(self):
        """int: Return current window height."""
        if self.window:
            return self.window.Height

    def close(self):
        """Close the window."""
        if self.window:
            self.window.Close()

    def close_others(self, all_open_outputs=False):
        """Close all other windows that belong to the current command.
```

(continues on next page)

(continued from previous page)

```
        Args:
            all_open_outputs (bool): Close all any other windows if True
        """
        if all_open_outputs:
            ScriptOutputManager.CloseActiveOutputWindows(self.window)
        else:
            ScriptOutputManager.CloseActiveOutputWindows(self.window,
                                                         self.output_id)

    def hide(self):
        """Hide the window."""
        if self.window:
            self.window.Hide()

    def show(self):
        """Show the window."""
        if self.window:
            self.window.Show()

    def lock_size(self):
        """Lock window size."""
        if self.window:
            self.window.LockSize()

    def save_contents(self, dest_file):
        """Save html code of the window.

        Args:
            dest_file (str): full path of the destination html file
        """
        if self.renderer:
            html = \
                self.renderer.Document.Body.OuterHtml.encode('ascii', 'ignore')
            doc_txt = self.renderer.DocumentText
            full_html = doc_txt.lower().replace('<body></body>', html)
            with open(dest_file, 'w') as output_file:
                output_file.write(full_html)

    def open_url(self, dest_url):
        """Open url page in output window.

        Args:
            dest_url (str): web url of the target page
        """
        if self.renderer:
            self.renderer.Navigate(dest_url, False)

    def open_page(self, dest_file):
        """Open html page in output window.

        Args:
            dest_file (str): full path of the target html file
        """
        self.show()
        self.open_url('file:///' + dest_file)

    def update_progress(self, cur_value, max_value):
```

(continues on next page)

(continued from previous page)

```
        """Activate and update the output window progress bar.

        Args:
            cur_value (float): current progress value e.g. 50
            max_value (float): total value e.g. 100

        Example:
            >>> output = pyrevit.output.get_output()
            >>> for i in range(100):
            >>>     output.update_progress(i, 100)
        """
        if self.window:
            self.window.UpdateProgressBar(cur_value, max_value)

    def reset_progress(self):
        """Reset output window progress bar to zero."""
        if self.window:
            self.window.UpdateProgressBar(0, 1)

    @staticmethod
    def emojize(md_str):
        """Replace emoji codes with emoji images and print.

        Args:
            md_str (str): string containing emoji code

        Example:
            >>> output = pyrevit.output.get_output()
            >>> output.emojize('Process completed. :thumbs_up:')
        """
        print(emoji.emojize(md_str), end="")

    @staticmethod
    def print_html(html_str):
        """Add the html code to the output window.

        Example:
            >>> output = pyrevit.output.get_output()
            >>> output.print_html('<strong>Title</strong>')
        """
        print(coreutils.prepare_html_str(emoji.emojize(html_str)), end="")

    @staticmethod
    def print_code(code_str):
        """Print code to the output window with special formatting.

        Example:
            >>> output = pyrevit.output.get_output()
            >>> output.print_code('value = 12')
        """
        code_div = '<div class="code">{}</div>'
        print(coreutils.prepare_html_str(
                code_div.format(
                    code_str.replace('    ', ' '*4))), end="")

    @staticmethod
    def print_md(md_str):
```

(continues on next page)

(continued from previous page)

```
        """Process markdown code and print to output window.

        Example:
            >>> output = pyrevit.output.get_output()
            >>> output.print_md('### Title')
        """
        tables_ext = 'pyrevit.coreutils.markdown.extensions.tables'
        markdown_html = markdown.markdown(md_str, extensions=[tables_ext])
        markdown_html = markdown_html.replace('\n', '').replace('\r', '')
        html_code = emoji.emojize(coreutils.prepare_html_str(markdown_html))
        print(html_code, end="")

    def print_table(self, table_data, columns=[], formats=[],
                    title='', last_line_style=''):
        """Print provided data in a table in output window.

        Args:
            table_data (:obj:`list` of iterables): 2D array of data
            title (str): table title
            columns (:obj:`list` str): list of column names
            formats (:obj:`list` str): column data formats
            last_line_style (str): css style of last row

        Example:
            >>> data = [
            ... ['row1', 'data', 'data', 80 ],
            ... ['row2', 'data', 'data', 45 ],
            ... ]
            >>> output.print_table(
            ... table_data=data,
            ... title="Example Table",
            ... columns=["Row Name", "Column 1", "Column 2", "Percentage"],
            ... formats=['', '', '', '{}%'],
            ... last_line_style='color:red;'
            ... )
        """
        if last_line_style:
            self.add_style('tr:last-child {{ {style} }}'
                           .format(style=last_line_style))

        zipper = itertools.izip_longest
        adjust_base_col = '|'
        adjust_extra_col = ':---|'
        base_col = '|'
        extra_col = '{data}|'

        # find max column count
        max_col = max([len(x) for x in table_data])

        header = ''
        if columns:
            header = base_col
            for idx, col_name in zipper(range(max_col), columns, fillvalue=''):
                header += extra_col.format(data=col_name)

            header += '\n'
```

(continues on next page)

(continued from previous page)

```
        justifier = adjust_base_col
        for idx in range(max_col):
            justifier += adjust_extra_col

        justifier += '\n'

        rows = ''
        for entry in table_data:
            row = base_col
            for idx, attrib, attr_format \
                    in zipper(range(max_col), entry, formats, fillvalue=''):
                if attr_format:
                    value = attr_format.format(attrib)
                else:
                    value = attrib
                row += extra_col.format(data=value)
            rows += row + '\n'

        table = header + justifier + rows
        self.print_md('### {title}'.format(title=title))
        self.print_md(table)

    def insert_divider(self):
        """Add horizontal rule to the output window."""
        self.print_md('-----')

    def next_page(self):
        """Add hidden next page tag to the output window.

        This is helpful to silently separate the output to multiple pages
        for better printing.
        """
        self.print_html('<div class="nextpage"></div><div> </div>')

    @staticmethod
    def linkify(element_ids, title=None):
        """Create clickable link for the provided element ids.

        This method, creates the link but does not print it directly.

        Args:
            element_ids (`ElementId`) or
            element_ids (:obj:`list` of `ElementId`): single or multiple ids
            title (str): tile of the link. defaults to list of element ids

        Example:
            >>> output = pyrevit.output.get_output()
            >>> for idx, elid in enumerate(element_ids):
            >>>     print('{}: {}'.format(idx+1, output.linkify(elid)))
        """
        return coreutils.prepare_html_str(
            linkmaker.make_link(element_ids, contents=title)
            )

    def make_chart(self):
        """:obj:`PyRevitOutputChart`: Return chart object."""
        return charts.PyRevitOutputChart(self)
```

(continues on next page)

(continued from previous page)

```
    def make_line_chart(self):
        """:obj:`PyRevitOutputChart`: Return line chart object."""
        return charts.PyRevitOutputChart(self, chart_type=charts.LINE_CHART)

    def make_stacked_chart(self):
        """:obj:`PyRevitOutputChart`: Return stacked chart object."""
        chart = charts.PyRevitOutputChart(self, chart_type=charts.LINE_CHART)
        chart.options.scales = {'yAxes': [{'stacked': True, }]}
        return chart

    def make_bar_chart(self):
        """:obj:`PyRevitOutputChart`: Return bar chart object."""
        return charts.PyRevitOutputChart(self, chart_type=charts.BAR_CHART)

    def make_radar_chart(self):
        """:obj:`PyRevitOutputChart`: Return radar chart object."""
        return charts.PyRevitOutputChart(self, chart_type=charts.RADAR_CHART)

    def make_polar_chart(self):
        """:obj:`PyRevitOutputChart`: Return polar chart object."""
        return charts.PyRevitOutputChart(self, chart_type=charts.POLAR_CHART)

    def make_pie_chart(self):
        """:obj:`PyRevitOutputChart`: Return pie chart object."""
        return charts.PyRevitOutputChart(self, chart_type=charts.PIE_CHART)

    def make_doughnut_chart(self):
        """:obj:`PyRevitOutputChart`: Return dougnut chart object."""
        return charts.PyRevitOutputChart(self,
                                         chart_type=charts.DOUGHNUT_CHART)

    def make_bubble_chart(self):
        """:obj:`PyRevitOutputChart`: Return bubble chart object."""
        return charts.PyRevitOutputChart(self, chart_type=charts.BUBBLE_CHART)


def get_output():
    """:obj:`pyrevit.output.PyRevitOutputWindow` : Return output window."""
    return PyRevitOutputWindow()
```

15.2 pyrevit.output.linkmaker

15.2.1 Documentation

Handle creation of output window helper links.

pyrevit.output.linkmaker.**make_link**(*element_ids*, *contents=None*)

Create link for given element ids.

This link is a special format link with revit:// scheme that is handled by the output window to select the provided element ids in current project. Scripts should not call this function directly. Creating clickable element links is handled by the output wrapper object through the `linkify()` method.

Example

```
>>> output = pyrevit.output.get_output()
>>> for idx, elid in enumerate(element_ids):
>>>     print('{}: {}'.format(idx+1, output.linkify(elid)))
```

15.2.2 Implementation

```
"""Handle creation of output window helper links."""

import json

from pyrevit import HOST_APP
from pyrevit import DB
from pyrevit.coreutils.logger import get_logger


logger = get_logger(__name__)


PROTOCOL_NAME = 'revit://outputhelpers?'

DEFAULT_LINK = '<a title="Click to select or show element" ' \
                  'class="elementlink" {}>{}</a>'


def make_link(element_ids, contents=None):
    """Create link for given element ids.

    This link is a special format link with revit:// scheme that is handled
    by the output window to select the provided element ids in current project.
    Scripts should not call this function directly. Creating clickable element
    links is handled by the output wrapper object through the :func:`linkify`
    method.

    Example:
        >>> output = pyrevit.output.get_output()
        >>> for idx, elid in enumerate(element_ids):
        >>>     print('{}: {}'.format(idx+1, output.linkify(elid)))
    """
    elementquery = []
    if isinstance(element_ids, list):
        strids = [str(x.IntegerValue) for x in element_ids]
    elif isinstance(element_ids, DB.ElementId):
        strids = [str(element_ids.IntegerValue)]

    for strid in strids:
        elementquery.append('element[]={}'.format(strid))

    reviturl = '&'.join(elementquery)
    linkname = ', '.join(strids)

    if len(reviturl) >= 2000:
        alertjs = 'alert(&quot;Url was too long and discarded!&quot;);'
        linkattrs = 'href="#" onClick="{}"'.format(alertjs)
```

(continues on next page)

(continued from previous page)

```
    else:
        linkattrs = 'href="{}{}{}"'.format(PROTOCOL_NAME,
                                           '&command=select&',
                                           reviturl)

    return DEFAULT_LINK.format(linkattrs, contents or linkname)
```

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W