extools Documentation

Release 0.9.20

Chris Binckly (2665093 Ontario Inc.)

Feb 24, 2023

Contents

CHAPTER 1

extools

1.1 extools.errors

exception extools.errors.**ExToolsError**(*trigger_exc=None*)

Bases: RuntimeError

A custom runtime exception for errors generated by ExTools.

Parameters **trigger_exc** (*Exception*) – the original exception that triggered this one.

1.2 extools.error_stack

Tools for working with the Sage Error Stack through Extender.

extools.error_stack.**consume_errors**()

Consume the message on the stack and get them as (Pri, Msg) pairs.

Returns [(pri, msg), (pri, msg), . . .]

Return type list

extools - Python tools for Orchid Extender

This package provides a number of utility functions designed to make working with Extender for Sage 300 easier.

A View-like object is any object which responds to .put, .get, .recordClear, .browse, .fetch, .update, .insert.

extools.**lines_in**(*ds*)

Generator that yields all records in a datasource.

Parameters **ds** – an instance of an accpac.UIDataSource

extools.**lines_in_view**(*viewid*)

Generator that yields all records in a view.

Parameters **viewid** – The identifier of the datastore (i.e. dsOEORDH)

extools.**open_and_fetch_row**(*viewid*)

Open the view and fetch the first row.

Parameters **view** – RotoID of the view to open and fetch from

Return type accpack.View or None

extools.**success**(**args*)

Check if the return values from view actions indicate success.

Extender view calls return 0 on success and non-zero on failure. Use this function to check whether a collection of view calls have been successful.

Parameters ***args** – any number of return values.

Return type bool

```
view = openView("EX0001")
rc = view.recordClear()
br = view.browse("")
fe = view.fetch()
if not success(rc, br, fe):
    showMessage("Failed to open EX0001 and seek to the first record.")
```

CHAPTER 2

extools.view

2.1 extools.view.errors

exception extools.view.errors.**ExViewComposeError**(*rotoid*, *compose_list=[]*, *action_return=None*, *trigger_exc=None*)

Bases: *extools.view.errors.ExViewError*

Error raised while composing ExViews.

Parameters

- **compose_list** (*list(str)*) – list of rotoids the ExView is being composed with.
- **action_return** (*int*) – return code from the View call.
- **trigger_exc** (*Exception*) – the exception that triggered this one.

exception extools.view.errors.**ExViewError**(*rotoid*, *action=''*, *action_return=None*, *fargs=[]*, *fkwargs={}*, *trigger_exc=None*)

Bases: *extools.errors.ExToolsError*

Generic error raised by an ExView.

Parameters

- **rotoid** (*str*) – RotoID of the ExView that raised.
- **action** (*string*) – ExView action that raised.
- **action_return** (*int*) – return code from the View call.
- **trigger_exc** (*Exception*) – the exception that triggered this one.

exception extools.view.errors.**ExViewFieldDoesNotExist**(*rotoid*, *field=None*, *order=None*, *action=None*, *action_return=None*, *trigger_exc=None*)

Bases: *extools.view.errors.ExViewError*

Error raised when a field does not exist in the view.

Parameters

- **field** (*str*) – field requested.
- **trigger_exc** (*Exception*) – the exception that triggered this one.

exception extools.view.errors.**ExViewIndexError**(*rotoid*, *kwargs=None*, *action='order'*, *action_return=None*, *trigger_exc=None*)

Bases: *extools.view.errors.ExViewError*

Error raised when an invalid index is provided.

Parameters

- **order** (*int*) – index order requested.
- **trigger_exc** (*Exception*) – the exception that triggered this one.

exception extools.view.errors.**ExViewInterpolationError**(*rotoid*, *format_string*, *field*, *root_view*, *seek_to*, *trigger_exc=None*, ***kwargs*)

Bases: *extools.view.errors.ExViewError*

Error raised when string interpolation using ExViews fails.

Parameters

- **rotoid** (*str*) – rotoid of the view failing interpolation.
- **format_string** (*str*) – string being interpolated
- **root_view** (*str*) – the root view from which all fields can be found.
- **seek_to** (*dict*) – seek to parameters for the first index of the root view.

exception extools.view.errors.**ExViewInvalidOrder**(*rotoid*, *order=None*, *action_return=None*, *trigger_exc=None*)

Bases: *extools.view.errors.ExViewError*

Error raised while setting the index order.

Parameters

- **order** (*int*) – order requested.
- **action_return** (*int*) – return code from the View call.
- **trigger_exc** (*Exception*) – the exception that triggered this one.

exception extools.view.errors.**ExViewOpenError**(*rotoid*, *action_return=None*, *trigger_exc=None*)

Bases: *extools.view.errors.ExViewError*

Error raised while opening ExView.

Parameters

- **rotoid** (*str*) – RotoID of the ExView that raised.
- **action_return** (*int*) – return code from the View call.
- **trigger_exc** (*Exception*) – the exception that triggered this one.

exception extools.view.errors.**ExViewRecordDoesNotExist**(*rotoid*, *action*, *action_return=None*, *trigger_exc=None*)

Bases: *extools.view.errors.ExViewError*

Error raised while opening ExView.

Parameters

- **rotoid** (*str*) – RotoID of the ExView that raised.
- **action_return** (*int*) – return code from the View call.
- **trigger_exc** (*Exception*) – the exception that triggered this one.

2.2 Self-composing views

Sage 300 uses the concept of *views* to provide a simplified interface for accessing data. The view layer sits between the user interfaces, like screens and imports, and the database. The views are where Sage implements and enforces business logic.

Extender provides access to views in Python through the `View` object. A view is usually backed by exactly one database table and provides access to a single record at a time. Sage allows views that share a unique identifier to be *composed* with one another.

When two views are composed, when the unique key field changes in one view, it is changed in all other composed views as well. This makes accessing related data much easier: no need to seek twice.

Consider the case where a script must set the optional field `MYFIELD` on a detail line to `HASX` if any of the lot numbers in an OE Order start with `X`. Working with Extender `View` objects and without composition:

```
# Open the order header and seek to the order
oe0520 = openView("OE0520")

if not oe0520:                          # If the open failed, return.
    return 1

rc = oe0520.recordClear()
if rc != 0:                             # If the record clear failed.
    return rc

o  = oe0520.order(1)
if o != 0:                              # If setting view order fails.
    return o

pu = oe0520.put("ORDNUMBER", "ORD453")
if pu != 0:                             # If setting the key field fails.
    return pu

r  = oe0520.read()
if r != 0:                              # If reading the record fails.
    return r

# Get the order unique ID to lookup the detail lines.
orduniq = oe0520.get("ORDUNIQ")

# Open the order details and seek to the beginning
oe0500 = openView("OE0500")
```

(continues on next page)

(continued from previous page)

```
if not oe0500:
    return 1

rc = oe0500.recordClear()
if rc != 0:
    return rc

br = oe0500.browse("", 1)
if br != 0:
    return br

# In preparation for reading lots and optional fields, open the views.
oe0507 = openView("OE0507")             # Order detail lots
oe0501 = openView("OE0501")             # Order detail optional fields
if not (oe0507 and oe0501):
    return 1

# For each detail line in the order
while(oe0500.fetch() == 0):
    # Seek the lot view to the lots for this line.
    linenum = oe0500.get("LINENUM")

    rc = oe0507.recordClear()
    if rc != 0:
        return rc

    br = oe0507.browse(
            'ORDUNIQ = "{}" AND LINENUM = "{}"'.format(orduniq, linenum))
    if br != 0:
        return br

    # Look at each lot associated with the line
    while(oe0507.fetch() == 0):

        # Get the lot number
        lotnumf = oe0507.get("LOTNUMF")

        # Check the condition
        if lotnumf and lotnumf.startswith("X"):

            # Try to read the optional field
            rc = oe0501.recordClear()
            if rc != 0:
                return rc

            # The index requires order, line, and optional field keys.
            po = oe0501.put("ORDUNIQ", orduniq)
            pl = oe0501.put("LINENUM", linenum)
            pf = oe0501.put("OPTFIELD", "MYFIELD")
            if po != 0 or pl != 0 or pf != 0:
                continue

            r = oe0501.read()
            if r != 0:
                # The field doesn't exist yet, create it.
                rg = oe0501.recordGenerate()
                if rg != 0:
```

(continues on next page)

(continued from previous page)

```
                    return rg

                po = oe0501.put("ORDUNIQ", orduniq)
                pl = oe0501.put("LINENUM", linenum)
                pf = oe0501.put("OPTFIELD", "MYFIELD")
                pv = oe0501.put("VALUE", "HASX")
                if po != 0 or pl != 0 or pf != 0 or pv != 0:
                    return 1

                ins = oe0501.insert()
                if ins != 0:
                    return ins
            else:
                # The field does exist, check it is correct.
                if oe0501.get("VALUE") != "HASX":

                    # It isn't set correctly, update it.
                    pv = oe0501.put("VALUE", "HASX")
                    if pv != 0:
                        return pv

                    up = oe0501.update()
                    if up != 0:
                        return up
```

It is a bit of a mouthful. Most of the code is opening, setting up, and seeking views. At every step, we need to search for the `orduniq` and/or `linenum`. This is where composition helps: it eliminates the a lot of the repetitive work by automatically filling in the unique keys for composed views.

Views can be composed at runtime. We can compose the views we are working with as they all share the `ORDUNIQ` key.

Let's try again but this time manually compose the views we need.

```
# Open the views
oe0520 = View("OE0520")      # Order Header
oe0500 = View("OE0500")      # Order Details
oe0507 = View("OE0507")      # Order Detail Lot Numbers
oe0501 = View("OE0501")      # Order Detail Optional Field
if not (oe0520 and oe0500 and oe0507 and oe0501):
    return 1

# Compose them all together.
c20 = oe0520.compose(oe0500, None, None, None, None, None)

# The arguments to compose can be a bit like a shell game...
c00 = oe0500.compose(oe0520, oe0501, None, None, None, oe0507)
c07 = oe0507.compose(oe0500)
c01 = oe0501.compose(oe0500)

# Make sure the composing was successful for all views
if c20 != 0 or c00 != 0 or c07 != 0 or c01 != 0:
    return 1

rc = oe0520.recordClear()
if rc != 0:                              # If the record clear failed.
    return rc
```

(continues on next page)

(continued from previous page)

```
o  = oe0520.order(1)
if o != 0:                              # If setting view order fails.
    return o

pu = oe0520.put("ORDNUMBER", "ORD453")
if pu != 0:                             # If setting the key field fails.
    return pu

r  = oe0520.read()
if r != 0:                              # If reading the record fails.
    return r

# Now a magical thing has happened, the detail view is ready to
# read the lines for this order.
rc = oe0500.recordClear()
if rc != 0:
    return rc

# For each detail line in the order
while(oe0500.fetch() == 0):
    # The fetch causes the optional field and lot views to filter out
    # all but records for the current line.

    rc = oe0507.recordClear()
    if rc != 0:
        return rc

    # Look at each lot associated with the line
    while(oe0507.fetch() == 0):
        # Get the lot number
        lotnumf = oe0507.get("LOTNUMF")

        # Check the condition
        if lotnumf and lotnumf.startswith("X"):

            # Because the ORDUNIQ and LINENUM are set implicitly through
            # composition, use the setOptionalField helper from the
            # Extender View class
            so = oe0501.setOptionalField("MYFIELD", "HASX")
            if not so:
                return so
```

That is much better. However, you need to know how to compose these things. There are thousands of views in Sage and not all views can be composed with all others. Composition is also uni-directional: the Order Headers view is composed with the Order Details but the Order Details must also be composed with the Order Header!

Consider the OE Order Header. In the example above, the header is only partially composed. It can be composed with up to five other views, each of which can be composed with many others. Fully composing the OE Header View involves opening and composing 13 other views, a total of 26 lines.

The *extools.view.ExView()* class is self-composing, so you don't need to worry about opening and checking the views or playing the shell game with compose arguments.

```
# ExViews use exceptions, wrap it all in a try and provide helpful
# output if an error is encountered.
try:
```

(continues on next page)

(continued from previous page)

```
    # Open the Order Header view
    oe0520 = ExView("OE0520")
    # Fully compose it, creating a new property for each related view
    oe0520.compose()
    # Set the order to search by ORDNUMBER
    oe0520.order(1)
    # Seek to the order we want
    oe0520.seek_to(ORDNUMBER="ORD453")

    # For each detail line in the order
    for line in oe0520.oe0500.lines():

        # For each lot in the detail line
        for lot in line.oe0507.lines():

            # Get the lot number
            lotnumf = lot.get("LOTNUMF")

            # Check the condition
            if lotnumf and lotnumf.startswith("X"):

                # Use the setOptionalField helper from the ExView class
                oe0501.setOptionalField("MYFIELD", "HASX")

except ExViewError as e:
    showMessage("Failed to set MYFIELD: {}".format(e))
    return 1
```

The call to *extools.view.ExView.compose()* introspects the view to find the other views that it can be composed with. It then opens them all with the correct indexing and composes them with one another. Each composed view is set as a property of the parent view so you can access them easily.

In the manual compose example, the views OE0500, OE0501, OE0507, and OE0520 were composed together. The "compose tree" for those views is:

```
OE0520
   |_ OE0500
        |_ OE0520
        |_ OE0501
        |     |_ OE0500
        |_ OE0507
              |_ OE0500
```

Composing the OE0520 ExView creates the following properties on the exview instance:

```
exview = ExView("OE0500")
exview.compose()

OE0520                      exview
   |_ OE0500                exview.oe0500
        |_ OE0520           exview.oe0500.oe0520 (back to self)
        |_ OE0501           exview.oe0500.oe0501
        |     |_ OE0500     exview.oe0500.oe0501.oe0500 (back to parent)
        |_ OE0507           exview.oe0500.oe0507
              |_ OE0500     exview.oe0500.oe0507.oe0500 (back to parent)
```

Note that because views are often composed bi-directionally, each composed view has a property that links back to its

parent.

2.3 Optional fields

Optional fields are one of the few things implemented consistently in the Sage 300 views. `ExView` instances take advantage of this, automatically creating the following helpers to manage the optional fields associated with an entry:

- *extools.view.ExView.create_optfield()*
- *extools.view.ExView.update_optfield()*
- *extools.view.ExView.get_optfield()*
- *extools.view.ExView.delete_optfield()*
- *extools.view.ExView.update_or_create_optfield()*
- *extools.view.ExView.seek_to_optfield()*
- *extools.view.ExView.has_optfield()*
- *extools.view.ExView.optfields*

When an `ExView` instance is created, it checks to see if it has any of the fields in the special constant *extools.view.ExView.OPTFIELD_VIEW_HINTS*. If so, the helper methods are automatically added to the class.

Optional fields are really custom properties of the object they are attached to, i.e. the Order Header optional fields are really properties of the Header itself - they just weren't included in Sage. `ExView` applies this idea to views that are *composed*, automatically adding the helpers to both the optional field and the attached object.

For example, when you open an optional field view (like `OE0522` Order Optional Fields), the helpers are automatically added to the instance.

```
try:
    # Open the Order Optional Fields view
    # Because this is an optional field view, helpers are added on init.
    oe0522 = ExView("OE0522")

    # If the optional field exists
    if oe0522.has_optfield("MYFIELD"):

        # Tell the user its value.
        showMessage("Myfield is set to: {}".format(
            oe0522.get_optfield("MYFIELD")))

except ExViewError as e:
    showMessage("Failed get optional MYFIELD: {}".format(e))
    return 1
```

Because the Order Optional Fields are really properties of the `OE0520` Order Header view, when you compose the header view the helpers are added to it as well. This provides a convenient, idiomatic, shortcut to managing an object's optional fields:

```
try:
    # Open the Order Header view and compose it
    oe0520 = ExView("OE0520")
    oe0520.compose()

    # If the optional field exists
```

(continues on next page)

(continued from previous page)

```
    if oe0520.has_optfield("MYFIELD"):

        # Tell the user its value.
        showMessageBox("Myfield is set to: {}".format(
            oe0520.get_optfield("MYFIELD")))

except ExViewError as e:
    showMessage("Failed get optional MYFIELD: {}".format(e))
    return 1
```

In the composed scenario, access from both the object and the composed optional field view are available and equivalent.

```
try:
    # Open the Order Header view
    oe0520 = ExView("OE0520")
    oe0520.compose()

    # This will always evaluate to True
    oe0520.has_optfield("MYFIELD") == oe0520.oe0522.has_optfield("MYFIELD")

    oe0520.create_optfield("F1", 1)

    if oe0520.oe0522.has_optfield("F1"):
        # True, we just created it through the header!
        ...
except ExViewError as e:
    showMessage("Failed ...")
    return 1
```

2.4 extools.view.exsql

The ExSql view performs SQL queries directly against the database. It can be used to perform high performance reads or writes that are otherwise not allowed by Sage's validation (caveat emptor).

class extools.view.exsql.**ExSql**

A class for working with the CS0120 view.

CSQL_VIEWID = 'CS0120'

get(*field*, *_type=-1*, *size=-1*, *precision=-1*)

Get a field from view.

Overrides the default get to skip verification that the field exists.

query(*query*)

Perform an SQL query and return the view.

Parameters **query** (*str*) – an SQL query to execute.

Returns view with the first result fetched.

Return type *ExView*

Raises extools.view.errors.ExSqlError, *extools.view.errors.ExViewError*

If you only need to execute a query, consider using a context manager like *extools.view.exsql.exsql()* or *extools.view.exsql.exsql_result()*.

```
try:
    exs = ExSql()
    result = exs.query("SELECT ITEM FROM OEORDD WHERE "
                       "ORDUNIQ = {} AND LINENUM = {}".format(
                            234634, 2))
    if exs.fetch():
        item = result.get("ITEM")
    else:
        # Handle record doesn't exist
except ExSqlError as e:
    # Handle an SQL fail
except ExViewError as e:
    # Handle a view layer fail
```

query_results(*query*)

Perform a query and yield the resulting records one at a time.

Parameters **query** (*str*) – an SQL query to execute.

Yields ExView

Returns None

Raises extools.view.errors.ExSqlError, *extools.view.errors.ExViewError*

classmethod record_count(*table*)

Get the total number of records from a table.

Parameters **table** – name of the table to count records in.

Returns record count

Return type int

Raises extools.view.errors.ExSqlError, *extools.view.errors.ExViewError*

extools.view.exsql.**columns_for_table**(*table*)

extools.view.exsql.**exsql**()

Open an ExSql view and yield it.

Yields ExSql

Return type None

Raises ExSqlError, ExViewError

```
try:
    with exsql() as exs:
        exs.query("SELECT ITEM FROM OEORDD WHERE "
                  "ORDUNIQ = {} AND LINENUM = {}".format(
                       234634, 2))
        exs.fetch()
        item = exs.get("ITEM")
except ExSqlError as e:
    # Handle an SQL fail
except ExViewError as e:
    # Handle a view layer fail
```

extools.view.exsql.**exsql_result**(*query*)

Open an ExSql view, executes a query, and yield the results.

Parameters **query** (*str*) – SQL query to execute.

Yields ExSql

Return type None

Raises extools.view.errors.ExSqlError, *extools.view.errors.ExViewError*

```
query = ("SELECT ITEM FROM OEORDD WHERE "
         "ORDUNIQ = {} AND LINENUM = {}".format(
                234634, 2))
try:
    with exsql_result(query) as res:
        item = res.get("ITEM")
except ExSqlError as e:
    # Handle an SQL fail
except ExViewError as e:
    # Handle a view layer fail
```

extools.view.exsql.**insert_optional_field**(*table*, *keys*, *user*, *org*, *field*, *value*)

extools.view.exsql.**sql_escape**(*term*)

Escape an SQL string for TSQL server (double quoted).

The following terms are escaped:

- \ -> \
- ' -> ''
- " -> "
- / -> \/

Parameters **term** (*str*) – the string to escape

Returns TSQL escaped string

Return type str

```
query = "SELECT * from TABLE WHERE SHOW = '{showname}'"
showname = sql_escape("Bob's Burgers") # -> "Bob''s Burgers"
try:
    with exsql_result(query.format(showname=showname)) as result:
        return result.get("NETWORK")
except ExViewRecordDoesNotExist:
    showMessageBox("No such show!")
except ExSqlError:
    showMessageBox("Table or field do not exist.")
except ExViewError:
    showMessageBox("An error occurred in the view.")
```

extools.view.exsql.**update_optional_field**(*table*, *keys*, *user*, *org*, *field*, *value*)

2.5 extools.view.query

The ExQuery class, modelled loosely on Django's QuerySet and JS semantics for method chaining, makes it easy to get at and manipulate the data you need.

Data can generally be accessed quickly through the view if the record you're searching for is indexed using the fields you're using to look. If you're off index, access through the view will require programmatic filtering and can be cumbersome. In these cases, using SQL through CS0120 is preferred.

ExQuery selects the right method based on the view and query terms in use. If the data can be quickly retrieved from the view, ExQuery retrieves data through the view. Otherwise, it will autmatically build an SQL query to retrieve the objects for you.

For example, can you iterate over all orders for a customer? Although CUSTOMER is a key field of OE0520 (OEORDH), you can't put to it in an empty view, so SQL is the right approach. ExQuery abstracts this away.

```
with exview("OE0520") as orders:
    results = orders.where(CUSTOMER='1200')
    for result in results:
        showMessageBox(result.ordnumber)
```

ExQueries can be refined by adding more terms, which are AND'd together. All orders for customer 1200 shipped to the warehouse:

```
with exview("OE0520") as orders:
    results = orders.where(CUSTOMER='1200', SHIPTO='WAREHS')
    for result in results:
        showMessageBox(result.ordnumber)
```

You can also limit the number of results, from a particular offset if required, and control the ordering. Get the 11 - 20 orders for customer 1200, ordered by item total descending:

```
# Pagination style example
with exview("OE0520") as orders:
    results = orders.where(CUSTOMER='1200').order_by('ITMINVTOT').limit(10).offset(10)
    for result in results:
        showMessageBox(result.ordnumber)
```

You can chain where clauses as well. Get the 11 - 20 orders for customer 1200 that shipped to the warehouse, ordered by the total item amount descending:

```
with exview("OE0520") as orders:
    results = orders.where(CUSTOMER='1200').order_by('ITMINVTOT').limit(10).offset(10)
    # I want the same query results but with a ship to filter
    results = results.where(SHIPTO='WAREHS')
    for result in results:
        showMessageBox(result.ordnumber)
```

The example above is not as inefficient as it may seem. The query is not executed until results are actually read. The database isn't accessed until the beginning of the for loop, when results is first read from. The first query, without the SHIPTO qualifier, is never executed.

What about cases where you may need an OR? Orders for customer 1200 or 1100? ExQuery results can be combined like sets. So you can get their union (|), intersection (&), or difference (-) between two results sets.

```
with exview("OE0520") as orders:
    c1_results = orders.where(CUSTOMER='1200')
```

(continues on next page)

(continued from previous page)

```
    c2_results = orders.where(CUSTOMER='1100')
    # find the union of the two result sets
    results = c1_results | c2_results
    for result in results:
        showMessageBox(result.ordnumber)
```

How about writing back? You can do that too. Set all orders for customer 1200 shipping to the warehouse on hold:

```
with exview("OE0520") as orders:
    results = orders.where(CUSTOMER='1200').where(SHIPTO='WAREHS')
    for result in results:
        result.update(ONHOLD=1)
```

Every result row in an ExQuery results maintains it's primary key. When you update a result, the update is made through the view, which is seeked directly to the correct record using the stored key.

You can treat an ExQuery like a list: index and slice it however you like:

```
with exview("OE0520") as orders:
    results = orders.where(CUSTOMER='1200').order_by("ORDTOTAL")
    # most expensive order outstanding is first:
    most_expensive = results[0]
    # least expensive is last
    least_expensive = results[-1]
    # Top ten orders
    top_ten = results[0:10]
```

2.6 extools.view.utils

Utilities for working with views and data.

extools.view.utils.customer.**customer_group_for**(*customer_id*)

Get the customer group for a given customer ID.

Parameters **customer_id** (*str*) – the customer ID to find the group for.

Returns customer group or an empty string.

Return type str

extools.view.utils.item.**category_for**(*itemno*)

Get the category for an item.

extools.view.utils.item.**item_optfield**(*itemno*, *optfield*)

Get an item's optional field value.

Parameters

- **itemno** (*str*) – the I/C item number.
- **optfield** (*str*) – the optional field name.

Returns value if set, else None

extools.view.utils.item.**unformat_itemno**(*itemno*)

Unformat an item number.

Parameters **itemno** (*str*) – formatted item number.

Returns item number with formatting characters removed.

extools.view.utils.item.**uoms_for**(*itemno*)
Get a list of all UOMs for an item. :param itemno: formatted item number. :type itemno: str :returns: ExQuery(ITEMNO=itemno)

extools.view.utils.item.**vendors_for**(*unfmtitemno*)
Get a list of all vendors.

The returned list is sparse: Vendor 1 at index 0, Vendor 2 at index 1, etc. Vendors that are not set will be None.

Parameters **unfmtitemno** (*str*) – Unformatted item number.

extools.view.utils.order.**order_from_quotes**(*customer*, *quotes*)
Create an order for a customer from a list of quotes.

Parameters

- **customer** (*str*) – customer number
- **quotes** (*list*) – list of quote numbers

Returns new order (not yet inserted)

Return type *ExView*("OE0520")

extools.view.utils.order.**orduniq_for**(*ordnumber*)
Get the Unique Order Key (ORDUNIQ) given an Order Number.

Parameters **ordnumber** (*str*) – Order number to get the unique key for.

Returns ORDUNIQ or 0.0

Return type float

extools.view.utils.shipment.**shiuniq_for**(*shinumber*)
Get the shipment uniquifier for a shipment number.

Parameters **shinumber** (*str*) – shipment number.

Returns shipment uniquifier or 0.0 is no such shipment

Return type float

ExView is a fully functional wrapper around the Extender View object that raises exceptions instead of providing non-zero returns on error for the methods defined in *extools.view.ExView.WRAP*.

It supports all the methods of the `Extender.View` class, along with many extra helpers.

On startup an `ExView` introspects the underlying Sage view to automatically determine:

- The view's composition tree
- The field names
- The allowed indexes and their key fields

Based on this information, the class automatically configures itself to:

- Self-compose on request (see *extools.view.ExView.compose()*)
- Validate orders and keys before errors are raised by Sage
- **Automatically add the correct helpers**
  - For detail views, the `.lines()`, `.lines_from(start, end)`, `.lines_where(key=value, key=value, ...)` generators and `newline()` helper.

- For optional field views, or any view composed with an optional field view, enable the `create_optfield`, `update_optfield`, `get_optfield`, `update_or_create_optfield`, `seek_to_optfield`, and `delete_optfield` helpers.

`extools.view.`**`exview`**(*rotoid*, *index=-1*, *seek_to={}*, *fetch=True*, *compose=False*)

Context manager to cleanly open and use an ExView.

Parameters

- **`rotoid`** (*`str`*) – the RotoID of the Sage view.
- **`index`** (*`int`*) – the index to open the view with.
- **`seek_to`** (*`dict`*|*`None`*) – field value mapping to seek to after opening. When set to an empty dictionary, seek to the first line in the view. If set to `None`, disable seek after opening.
- **`fetch`** (*`bool`*) – automatically fetch the first matched record?
- **`compose`** (*`bool`*) – automatically compose before seeking?

Raises ExViewError

Return type None

When called the context manager will yield an open view object. On exit of the block the view will be closed cleanly.

```
with exview("EX0001") as view:
    try:
        view.recordClear()
        view.browse("")
        view.fetch()
        value = view.get("KEY")
    except ExViewError as err:
        showMessageBox("Failed to get KEY, {}.".format(err))
```

`extools.view.`**`exgen`**(*rotoid*, *index=-1*, *seek_to={}*)

Generator for iterating over all the entries in a view.

Parameters

- **`rotoid`** (*`str`*) – the RotoID of the Sage view.
- **`index`** (*`int`*) – the index to open the view with.
- **`seek_to`** (*`dict`*|*`None`*) – field value mapping to seek to after opening. When set to an empty dictionary, seek to the first line in the view. If set to `None`, disable seek after opening.

Raises ExViewError

Return type None

When called, the generator will seek the view to the requested records, or the first record if `seek_to` is empty. It will then yield all matching rows and then cleanly close the view.

```
for record in exgen("EX0001"):
    try:
        record.get("FIELD")
    except ExViewError as err:
        showMessageBox("Failed to get FIELD, {}.".format(err))
```

`extools.view.`**`EXVIEW_BLACKLIST = {'OE0999'}`**

Views that can never be composed with any other.

class extools.view.**ExView**(*rotoid*, *index=-1*, *seek_to={}*, *native_types=False*, *fetch=True*, *_root=True*, *_me=None*, *_cviews=[]*)

An exception raising wrapper around the Extender View class.

ExViews can be used to replace repetitive error checking and to take advantage of the try/except/else/finally construct in Python.

Parameters

- **rotoid** (*str.*) – the RotoID of the Sage view.
- **index** (*int.*) – the index to open the view with.
- **seek_to** (*dict|None*) – field value mapping to seek to after opening. When set to an empty dictionary, seek to the first line in the view. If set to None, disable seek after opening.

Raises ExViewError

Return type *ExView*

Replace this:

```
view = openView("EX0001")
if not view:
    showMessageBox("Failed to open view.")
    return 1
rc = view.recordClear()
if rc != 0:
    showMessageBox("Failed to record clear.")
    return 1
br = view.browse("")
if br != 0:
    showMessageBox("Failed to browse.")
    return 1
fe = view.fetch()
if fe != 0:
    showMessageBox("Failed to fetch.")
    return 1

value = view.get("KEY")

if view:
    view.close()
```

With this:

```
try:
    view = ExView("EX0001")
    value = view.get("KEY")
except ExViewError as err:
    showMessageBox("Failed to get KEY, {}.".format(err))
    return 1
finally:
    view.close()
```

You can even include the traceback using the ExMessages:

```
try:
    view = ExView("EX0001")
    value = view.get("KEY")
except ExViewError as err:
```

(continues on next page)

(continued from previous page)

```
    # Use ExMessages to display an error level message box and
    # log to a file (if configured). The last exception traceback
    # will be included in both the box and log if ``exc_info=True``.
    exm.error("Failed to get KEY, {}.".format(err), exc_info=True)
    return 1
finally:
    view.close()
```

ExViews can also self-compose, composing the view and all its related views automatically. Fully composed views require more database operations every time the header is changed and do not perform as well as standalone views or SQL access. However, in cases where performance isn't paramount you cannot beat the convenience.

```
from extools import success
from extools.view import ExView
from extools.message import ExMessages

exm = ExMessages("compose-test", ExMessages.DEBUG)

try:
    exv = ExView("OE0520")
    exv.compose()
except Exception as e:
    exm.error("Failed to setup view: {}".format(e), exc_info=True)

# Seek to order ORD000000000064
try:
    # Use index 1, key (ORDNUMBER, )
    exv.order(1)
    exv.seek_to(ORDNUMBER="ORD000000000064")
except Exception as e:
    exm.error("Failed to seek: {}".format(e), exc_info=True)

# Perform an action on each of the detail lines in the order
try:
    for line in exv.oe0500.lines():
        exm.info("Read new line {}".format(line.get("ITEM")))
        # perform many important actions...
except Exception as e:
    exm.error("Failed to perform action: {}".format(e), exc_info=True)
```

lines(*self*)

A generator that yields all lines in a detail view.

Only available on detail views.

Return type None

Yields ExView

```
for line in oe500.lines():
    # line now contains the oe0500 view seeked to the next line
```

lines_from(*start*, *end=None*)

A generator that yields all lines from `start` to `end`.

Only available on detail views.

Parameters

- **start** (*int*) – line to start at (numbering starts at 0)
- **end** (*int*) – line to end on (inclusive)

Yields ExView

Return type None

```
for line in oe500.lines_from(2, 3):
    # line now contains the oe0500 view seeked to the second line
    # there will be one more iteration with the third line
```

lines_where(*key=value*, *key=value*, *key=value*, ...)

A generator that yields all lines matched by browsing for the provided keys. All key value pairs are combined using the AND condition and used to browse.

Only available on detail views.

Parameters

- **key** (*str*) – a field name in the view.
- **value** (*any*) – the value to browse to

Yields ExView

Return type None

```
for line in oe500.lines_where(ORDUNIQ=234234):
    # line now contains the oe0500 view seeked to first line or
    # the oder with unqiue key 234234.
```

create_optfield(*field*, *value*)

Create a new optional field, set its value, and save it.

Parameters

- **field** (*str*) – Optional field name.
- **value** (*builtins.**) – Optional field value.

Return type None

Raises ExViewError

```
try:
    oe0500 = ExView("OE0500")
    oe0500.compose()
    # When the view is composed, the associated optional field view
    # OE0522 is auto-detected so you can call the ``*_optfield``
    # methods directly on oe0500.
    oe0500.create_optfield("MYFIELD", "NEWVAL")
except ExViewError as e:
    # Do something on fail.
```

update_optfield(*field*, *value*)

Update an existing optional field.

Parameters

- **field** (*str*) – Optional field name.
- **value** (*builtins.**) – Optional field value.

Return type None

Raises ExViewError

```
try:
    oe0500 = ExView("OE0500")
    oe0500.compose()
    # When the view is composed, the associated optional field view
    # OE0522 is auto-detected so you can call the ``*_optfield``
    # methods directly on oe0500.
    oe0500.update_optfield("MYFIELD", "UPDATEDVAL")

    # The composed OE0522 view is also accessible.
    oe0500.oe0522.update_optfield("MYFIELD", "UP2DATEVAL")

except ExViewError as e:
    # Do something on fail.
```

delete_optfield(*field*)

Delete an existing optional field.

Parameters **field** (*str*) – Optional field name.

Return type None

Raises ExViewError

```
try:
    oe0500 = ExView("OE0500")
    oe0500.compose()

    # When the view is composed, the associated optional field view
    # OE0522 is auto-detected so you can call the ``*_optfield``
    # methods directly on oe0500.
    oe0500.delete_optfield("MYFIELD")

    # The composed OE0522 view is also accessible.
    oe0500.oe0522.delete_optfield("MYFIELD")

except ExViewError as e:
    # Do something on fail.
```

get_optfield(*field*)

Get the value of an existing optional field.

Parameters **field** (*str*) – Optional field name.

Returns Optional field value.

Return type builtins.*

Raises ExViewError

```
try:
    oe0500 = ExView("OE0500")
    oe0500.compose()

    # When the view is composed, the associated optional field view
    # OE0522 is auto-detected so you can call the ``*_optfield``
    # methods directly on oe0500.
    value = oe0500.get_optfield("MYFIELD")
```

(continues on next page)

(continued from previous page)

```
    # The composed OE0522 view is also accessible.
    value = oe0500.oe0522.get_optfield("MYFIELD")

except ExViewError as e:
    # Do something on fail.
```

seek_to_optfield(*field*)

Seek the view to an existing optional field.

Parameters **field** (*str*) – Optional field name.

Return type None

Raises ExViewError

```
try:
    oe0500 = ExView("OE0500")
    oe0500.compose()

    # When the view is composed, the associated optional field view
    # OE0522 is auto-detected so you can call the ``*_optfield``
    # methods directly on oe0500.
    oe0500.seek_to_optfield("MYFIELD")

    # The composed OE0522 view is also accessible.
    # Now that is has seeked to MYFIELD extract the value with get.
    value = oe0500.oe0522.get("VALUE")

except ExViewError as e:
    # Do something on fail.
```

update_or_create_optfield(*field*, *value*)

Update an existing optional field if it exists, otherwise create it.

Parameters

- **field** (*str*) – Optional field name.
- **value** (*builtins.**) – Optional field value.

Return type None

Raises ExViewError

```
try:
    oe0500 = ExView("OE0500")
    oe0500.compose()
    # When the view is composed, the associated optional field view
    # OE0522 is auto-detected so you can call the ``*_optfield``
    # methods directly on oe0500.
    oe0500.update_or_create_optfield("MYFIELD", "UPDATEDVAL")

    # The composed OE0522 view is also accessible.
    oe0500.oe0522.update_or_create_optfield("MYFIELD", "UPDATEVAL")

except ExViewError as e:
    # Do something on fail.
```

has_optfield(*field*)

Check if an optional field exists.

Parameters **field** (*str*) – Optional field name.

Returns True if an optional field with name `field` exists.

Return type bool

Raises ExViewError

```
try:
    oe0500 = ExView("OE0500")
    oe0500.compose()
    # When the view is composed, the associated optional field view
    # OE0522 is auto-detected so you can call the ``*_optfield``
    # methods directly on oe0500.
    if oe0500.has_optfield("MYFIELD"):
        showMessageBox("MYFIELD already defined.")

    # The composed OE0522 view is also accessible.
    oe0500.oe0522.has_optfield("MYFIELD", "UPDATEVAL")

except ExViewError as e:
    # Do something on fail.
```

optfields

Get all optional fields for a view.

Returns mapping of field names to values.

Return type dict

Raises ExViewError

```
try:
    oe0500 = ExView("OE0500")
    oe0500.compose()

    # oe0500.optfields is now poplated with all the optional
    # fields currently defined for the header.
    # { "FIELDNAME": "VALUE", "F1": 1, "MYFIELD": "VALUE"}
    if "MYFIELD" in oe0500.optfields.keys():
        showMessage("MYFIELD is set to {}".format(
            oe0500.optfields["MYFIELD"]))
except ExViewError as e:
    # Do something on fail.
```

ATTRS = {'A': 2, 'C': 16, 'E': 4, 'K': 8, 'P': 32, 'R': 48, 'X': 64}

ATTR_A = 2

ATTR_COMPUTED = 16

ATTR_EDITABLE = 4

ATTR_KEY = 8

ATTR_P = 32

ATTR_R = 48

ATTR_X = 64

DETAIL_VIEW_HINTS = {'CNTENTR', 'DETAILNUM', 'ENTRY', 'LINENUM'}

Views containing any one of these fields may be detail views.

OPTFIELD_VIEW_HINTS = {'OPTFIELD'}
Views containing any one of these fields may be optional field views.

WRAP = ['fetchLock', 'readLock', 'insert', 'delete', 'init', 'post', 'process', 'verif
These View functions raise an `ExViewError` on non-zero return.

all(*ascending=True*)
Generator that yields once for each record in the view.

Raises ExViewError

Yields ExView

cached_view(**args*, ***kwargs*)

close(**args*, ***kwargs*)

compose(**args*, ***kwargs*)

compose_from(*composed_views*)

copy_to(*view2*, *force=True*, *exclude=[]*, *include=[]*, *post_process=[]*, *skip_keys=True*, *skip_computed=True*, *save=False*)
Copy the current object to view2.

Parameters

- **view2** (ExView) – the view to copy to.
- **exclude** (*str[]*) – Fields to exclude from copy.
- **include** (*str[]*) – Fields to include, excluding all others.
- **post_process** (*int[]*) – run process with these processcmds after copy.
- **skip_keys** (*bool*) – skip fields with the Key attribute. Default: yes.
- **skip_computed** (*bool*) – skip fields with the Key attribute. Default: yes.
- **save** (*bool*) – insert the object after copy. Default: no.

Raises ***ExViewError*** – when any error occurs during the copy.

create(***fields*)
Generate and insert a new entry with field/value pairs.

Parameters **fields** (*field=value*) – field value pairs that will be set on the new entry.

Return type None

Raises ExViewError

current_key()
Get the current unique key identifying the view record.

Returns {field: value, field: value. . . }

exists()
Wrap exists to return True or False and not raise.

Returns True if record in view exists (has been added), else False

Return type bool

fetch()
A special wrapper because a non-zero fetch return isn't an error.

Returns True if a new line was fetched, else False.

Return type bool

classmethod from_me(*_me*)

get(*field*, *_type=-1*, *size=-1*, *precision=-1*, *verify=True*)
A special wrapper because get doesn't return 0 on success.

Parameters

- **field** (*str*) – field name to get.
- **verify** – verify that the field is listed in the view fields?

Type bool

Returns value in the view.

Return type builtin.*

Raises ExViewFieldDoesNotExist

has_optfield_view
Is this view composed with an optional field view?

Returns True if this view is composed with an optional field view.

Return type bool

is_optfield_view
Is this an optional field view?

Returns True if this view is an optional field view.

Return type bool

order(*_ord*)
Wrap the order to track state in the class as it can't be queried.

Parameters **index** (*int*) – the index ID to order by.

Return type None

Raises ExViewError

parent_key()
Get the current unique key identifying the view record's parent.

Only relevant for detail views, return the key components before the last one.

The views, as classified by Sage, may either be header, detail, flat or batch. Both detail, and headers with composite keys, may be enumerated.

Returns {field: value, field: value...}

read()
A special wrapper to raise ExViewRecordDoesNotExist.

Raises ExViewRecordDoesNotExist

remove_cached_view(**args*, ***kwargs*)

seek_to(*fetch=True*, ***kwargs*)
Intelligently seek to a specific entry.

This seek to implementation accepts an arbitrary set of field value pairs and then seeks to the entry using one of three methods:

- If the current View order index is made up of exactly the fields requested, perform a straight put and read.
- If the current View has an index made up of exactly the fields requested, temporarily change the index and perform and put a read.
- If the current View does not have and index made up of exactly the fields requested, attempt to browse and fetch the record.

Parameters

- **fetch** (*bool*) – fetch after seeking? Default to true.
- **kwargs** (*dict*) – (key)=(value) pairs, where the keys must be the same as the current index keys.

Return type None

Raises ExViewError

```
viewid = "OE0500"
try:
    exv = ExView(viewid) # Open Order Details, default view order 0

    # Seek to the 7th line of the order with unique key 1024
    # The default view order is 0: (ORDUNIQ, LINENUM, )
    exv.seek_to(ORDUNIQ=1024, LINENUM=7)

    # Get details from the record and process or update.
    item = exv.get("ITEM")
    ...
except ExViewError as e:

    # The error, "failed to [open|seek]", is contained in the
    # error message.
    showMessage("Error doing something with view {}: {}".format(
        viewid, e))
```

classmethod table_name(*rotoid*)

to_dict()

Return all the fields in a view as a dictionary.

Useful for caching full rows for later use.

update(***fields*)

Update an entry with field/value pairs.

Parameters **kwargs** (*field=value*) – field value pairs that will be set on the new entry.

Return type None

Raises ExViewError

where(***criteria*)

Get an ExQuery to retrieve records with the given criteria.

Parameters **criteria** – field=value criteria to browse to.

Returns ExQuery

Raises ExViewError

CHAPTER 3

extools.message

class extools.message.**ExMessages**(*name*, *level=None*, *log_path=None*, *programs=[]*, *box=True*, *disabled=False*, *key=''*, *handler=None*)

Bases: object

A logger like object for writing messages for the user.

The ExtenderMessageWriter acts like a logger, allowing a developer to add messages that are only displayed to the user if the current level is greater than or equal to the message level being called.

Messages at debug and below, as well as those at error or above, support displaying the last exception traceback to make debugging easier.

Parameters

- **name** (*str*) – the name to log under.
- **level** (*int*) – the level at and below which to display messages.
- **log_path** (*str*) – the path of a log file to write to.
- **programs** (*list*) – the list of programs for which to display messages. For example, if programs were ["OE1100",] then messages will only be displayed if the Order Entry program is currently running.
- **box** (*True (showMessageBox), False (message stack), None (suppress)*) – indicates whether to show a message box, add a message to the Sage message stack, or suppress UI messages. Defaults to True.
- **disabled** (*bool*) – disable all messages and logging. Defaults to False.

CRITICAL = 1

DEBUG = 20

ERROR = 5

INFO = 15

LEVELS = (0, 1, 5, 10, 15, 20, 25)

Supported log levels in decreasing order of severity.

PANIC = 0

RAW = 25

WARNING = 10

YES_NO_DIALOG = 4

YES_NO_DIALOG_NO = 7

YES_NO_DIALOG_YES = 6

crit(*msg*, *exc_info=None*)
Display and log a critical message.

Parameters

- **msg** (*str*) – message to write.
- **exc_info** (*bool*) – include last exception backtrace?

Return type None

debug(*msg*, *exc_info=False*)
Display and log a debug message.

Parameters

- **msg** (*str*) – message to write.
- **exc_info** (*bool*) – include last exception backtrace?

Return type None

debug_error_stack()
Write the contents of the error stack to log as debug messages and clear the stack.

error(*msg*, *exc_info=None*)
Display and log an error message.

Parameters

- **msg** (*str*) – message to write.
- **exc_info** (*bool*) – include last exception backtrace?

Return type None

info(*msg*)
Display and log an info message.

Parameters **msg** (*str*) – message to write.

Return type None

panic(*msg*, *exc_info=None*)
Display and log a panic message.

Parameters

- **msg** (*str*) – message to write.
- **exc_info** (*bool*) – include last exception backtrace?

Return type None

classmethod prompt(*title*, *message*)
Display a Yes/No dialog prompt.

Parameters

- **title** (*str*) – The message box title.
- **message** (*str*) – The prompt to display.

Returns True if User selects Yes, else No

Return type bool

raw(*msg*, *exc_info=False*)
Display and log raw output.

Parameters

- **msg** (*str*) – message to write.
- **exc_info** (*bool*) – include last exception backtrace?

Return type None

warn(*msg*)
Display and log a warning message.

Parameters **msg** (*str*) – message to write.

Return type None

extools.message.**logger_for_module**(*module_name*, *level=None*, *box=None*, ***kwargs*)

CHAPTER 4

extools.test

class extools.test.**ExTestCase**(*log_level=15*)

Bases: object

A self running test case class for Extender scripts.

ExTestCase can be used to test code in the Extender environment. Test cases will run with access to the current company. Write your tests against the sample data in SAMINC, setup anything else you need on the fly, to make it easy to build a repeatable test environment.

Parameters **log_level** (*int*) – level of the built-in logger.

To create your own tests:

1. subclass ExTestCase
2. define the `.setup()` and `.teardown` methods
3. create as many methods starting with `test_` as you'd like
4. run your test suite by creating an instance and calling `.run()`

Run all the tests in a project using the included ExTestRunner module.

Let's say you want to test your set order decription custom function, which sets the order decription to the customer number.

```
from extools.view import exview

def set_description_to_customer_number(ordnumber):
    '''Set the order description to the customer number.

    :param ordnumber: the order number to update.
    :type ordnumber: str
    :rtype: None
    :raises: ExViewError
    '''
    with exview("OE0500", seek_to={"ORDNUMBER": ordnumber}) as exv:
        exv.update(DESC=exv.get("CUSTOMER"))
```

Using the *extools.view.exview()* context manager makes opening, closing, and seeking the view easy. To test it, we will need a record in the SAMINC database that we can change the Description on. The first, ORD000000000001, seems to fit the bill.

```
from extools.test import ExTestCase
from mymodule import set_description_to_customer_number

class MyTest(ExTestCase):

    # Make the order to work on a constant
    ORDER_NUMBER = "ORD000000000001"
    ORDER_VIEW = "OE0500"

    def setup(self):
        # Make sure the field isn't already set to the customer number
        with exview(ORDER_VIEW, seek_to={"ORDNUMBER": ORDER_NUMBER}) as exv:
            if exv.get("DESC") == exv.get("CUSTOMER"):
                exv.update(DESC="Description")

    # The test method must start with ``test_`` to be auto-detected.
    def test_set_description_to_customer_number(self):
        # Use the built-in assertions to check the pre-
        with exview(ORDER_VIEW, seek_to={"ORDNUMBER": ORDER_NUMBER}) as exv:
            self.assertTrue(not exv.get("CUSTOMER") == exv.get("DESC"))

        set_description_to_customer_number(ORDER_NUMBER)

        #and post conditions
        with exview(ORDER_VIEW, seek_to={"ORDNUMBER": ORDER_NUMBER}) as exv:
            self.assertTrue(exv.get("CUSTOMER") == exv.get("DESC"))

def main():
    # To run your tests, instantiate the class and run it!
    mt = MyTest()
    mt.run()
```

INDEX_MAX = 99

assert_equal(**args*, ***kw*)

assert_raises(**args*, ***kw*)

assert_true(**args*, ***kw*)

generate_index()
Get the next available view index and increment the counter.

run(*with_transaction=True*)
Run all the tests in the class and provide a report.

setup()
Steps that are run before every test.

setup_class()
Steps that are run once before the test suite starts.

teardown()
Steps that are run after every test.

teardown_class()
Steps that are run once after the test suite ends.

exception extools.test.**ExTestError**(*trigger_exc=None*)

Bases: *extools.errors.ExToolsError*

Raised by failed test cases.

extools.test.**asserts**(*method*)

CHAPTER 5

extools.ui

5.1 extools.ui.bare

class extools.ui.bare.**BareUI**
Bases: object

Am empty UI for running scripts without displayinga window.

extools.ui.bare.**bareui**(*close=True*)
Get a BareUI instance temporarily.

Useful for running scripts that pop up messages but don't need to leave the UI window lingering.

Parameters **close** (*bool*) – whether to automatically close the UI. Disable to capture lingering errors.

```
# myextenderscript.py
from accpac import *
from extools.exui.bare import bareui

def main(*args, **kwargs):
    with bareui():
        showMessageBox("This will be displayed!")
        # ...
```

5.2 Grid Column UIs

Some pre-built classes are included for quickly customizing a grid on screen.

The *extools.ui.callback_column.CallbackColumnUI* adds a column to the grid that is populated with a callback. It looks after the screen and column setup, and calls the callback to popluate the cell whenever a value is requested. The callback receives the grid edit event and data source as arguments.

The *extools.ui.optfield_column.OptfieldColumnUI* adds a column to the grid that is populated with an optional field. The UI supports both editable and non-editable cells and automatically writes back values to the optional field data source.

The extools.ui.optfield_column.OptfieldMultiColumnUI adds columns to the grid that are populated optional fields. The UI supports both editable and non-editable cells and automatically writes back values to the optional field data source.

The *extools.ui.datasource_column.DatasourceColumnUI* adds a column that is populated with a datasource field - any datasource, it need not be the one that backs the grid.

class extools.ui.callback_column.**CallbackColumnUI**(*ds*, *grid_control_name*, *fields*)

Callback Column UI adds a column to a grid populated with a callback.

Parameters

- **ds** (*str*) – Data source name.
- **grid_control_name** (*str*) – Name of the grid control to add a column to.
- **fields** (*[{'caption': str, 'hijack': str, 'callback': function}, ..]*) – a list of dictionaries containing the field definitions.

To use the CallbackColumn UI, create a new Screen script that instantiates an instance of the class, customizing it with the arguments.

Below is an example of adding a column, *Qty in Ea.* to the O/E Order Entry grid, that is populated with the quantity ordered in eaches, regardless of the line's Unit of Measure.

```
# OE1100
from accpac import *
from extools.ui.callback_column import CallbackColumnUI

DATASOURCE = "adsOEORDD"
GRID_CONTROL = "avlOEORDDdetail1"

def quantity_in_eaches_callback(event, datasource):
    qty = datasource.get("QTYORDERED")
    if qty:
        return qty * datasource.get("UNITCONV")
    return 0

fields = [{
    'caption': "Qty in Ea.",
    'hijack': 'ITEM',
    'callback': quantity_in_eaches_callback
},]

def main(*args, **kwargs):
    CallbackColumnUI(DATASOURCE, GRID_CONTROL, fields)
```

class extools.ui.optfield_column.**OptfieldColumnUI**(*caption*, *optional_field*, *optf_datasource*, *grid_control_name*, *hijack='ITEM'*, *default=''*, *editable=False*)

Optfield Column UI adds a column populated with a optional field value.

Parameters

- **caption** (*str*) – Column caption.

- **optional_field** (*str*) – the name of the data source field.
- **optf_datasource** (*str*) – Optional field datasource name.
- **grid_control_name** (*str*) – Name of the grid control to add a column to.
- **hijack** (*str*) – Existing grid column to hijack.
- **default** (*object*) – Default value for the field.
- **editable** (*bool*) – Allow field to be edited?

To use the OptfieldColumn UI, create a new Screen script that initializes an instance of the class, customizing it with the arguments.

Below is an example of adding a column, *Warranty*, to the O/E Order Entry grid, that is populated with the WARRANTY optional field for the line.

```
# OE1100
from accpac import *
from extools.ui.optfield_column import OptfieldColumnUI

CAPTION = "Warranty"
OPTFIELD = "WARRANTY"
OPTF_DATASOURCE = "dsOEORDDO"
GRID_CONTROL = "avlOEORDDdetail1"
HIJACK_COLUMN = "ITEM"

def main(*args, **kwargs):
    OptfieldColumnUI(CAPTION, OPTFIELD, OPTF_DATASOURCE, GRID_CONTROL,
                     HIJACK, default="", editable=True)
```

class extools.ui.optfield_multicolumn.**OptfieldMultiColumnUI**(*optf_datasource*, *grid_control_name*, *columns*)

Optfield Column UI adds a column populated with a optional field value.

Parameters

- **optf_datasource** (*str*) – Optional field datasource name.
- **grid_control_name** (*str*) – Name of the grid control to add a column to.
- **columns** (*dict*) – Columns to create

To use the OptfieldMultiColumn UI, create a new Screen script that initializes an instance of the class, passing the column configuration as a dictionary.

Below is an example of adding two columns, *Warranty* and *Warranty Period*, to the O/E Order Entry grid, that is populated with the WARRANTY optional field for the line.

```
# OE1100
from accpac import *
from extools.ui.optfield_column import OptfieldMultiColumnUI

COLUMNS = {
    "Warranty": {
        'optional_field': 'WARRANTY',
        'default': False,
        'editable': False,
        'hijack': 'LOCATION'
    },
```

(continues on next page)

(continued from previous page)

```
    "Warranty Period": {
        'optional_field': 'WARRANTYPRD',
        'default': "60 Days",
        'editable': True,
        'hijack': 'ITEM',
    },

OPTF_DATASOURCE = "dsOEORDDO"
GRID_CONTROL = "avlOEORDDdetail1"

def main(*args, **kwargs):
    OptfieldMultiColumnUI(OPTF_DATASOURCE, GRID_CONTROL, COLUMNS)
```

class extools.ui.datasource_column.**DatasourceColumnUI**(*caption*, *ds*, *grid_control_name*, *hijack*, *gettext_callback*)

Datasource Column UI adds a column populated with a datasource field.

The datasource does not need to be the same datasource as

Parameters

- **caption** (*str*) – Column caption.
- **grid_ds** – Data source name.
- **grid_control_name** (*str*) – Name of the grid control to add a column to.
- **hijack** (*str*) – Existing grid column to hijack.
- **field** (*str*) – the name of the data source field.

To use the DatasourceColumn UI, create a new Screen script that initializes an instance of the class, customizing it with the arguments.

Below is an example of adding a column, *Unit Conversion* to the O/E Order Entry grid, that is populated with the unit conversion for the line.

```
# OE1100
from accpac import *
from extools.ui.datasource_column import DatasourceColumnUI

CAPTION = "Unit Conversion"
DATASOURCE = "adsOEORDD"
GRID_CONTROL = "avlOEORDDdetail1"
HIJACK_COLUMN = "ITEM"
FIELD = "UNITCONV"

def main(*args, **kwargs):
    DatasourceColumnUI(CAPTION, DATASOURCE, GRID_CONTROL,
                       HIJACK, FIELD)
```

5.3 extools.ui.field_security

Field Security UIs

UIs for implementing additional security at the field level.

Use the *extools.ui.field_security.permissions_callback_for_table()* together with the *extools.ui.field_security.FieldSecurityCallbackUI* to quickly apply permissions to pretty much any field on a Sage screen.

```
# AP1200
# Add permissions to AP Vendor Tax Number field

MY_PERMISSIONS_TABLE = "MYMODULE.MYPERMS"
'''Table format
| USER | CANREAD | CANWRITE |'''

# The tax number field control name from accpacUIInfo
TAX_NUM_CONTROL = "afecAPVENtaxnbr"
# The tab control name
TAB_CONTROL = "SSTab1"
# The id of the "Invoicing" tab, which has the field.
TAB_ID = 1

def main(*args, **kwargs):
    callback = permissions_callback_for_table(
            MY_PERMISSIONS_TABLE, "USER", "CANREAD", "CANWRITE")
    ui = FieldSecurityCallbackUI(
            TAX_NUM_CONTROL, callback, TAB_CONTROL, TAB_ID)
```

class extools.ui.field_security.**FieldSecurityCallbackUI**(*control*, *callback*, *tab_control=''*, *tab_id=None*, *ds_name=''*)

Bases: object

Apply custom permissions to a specific on-screen field.

Parameters

- **control** (*str*) – name of the on-screen control.
- **callback** (*func*) – callback function to check perms. receives (user, control, ds). callback must return (bool(can_read), bool(can_write)).
- **tab_control** (*str*) – name of the tab control (if applicable).
- **tab_id** (*str*) – tab id on which the field appears (if applicable).
- **ds_name** (*str*) – name of a ds (i.e. adsOEORDH) to open and pass to callback.

onControlInfo(*info*)

onTabClick(*tab*)

On tab change hide or show field based on callback.

extools.ui.field_security.**permissions_callback_for_table**(*view*, *user_field*, *read_field*, *write_field*)

Generate a callback that checks a user's permissions in a view.

Best used with the *FieldSecurityCallbackUI*, it will check a table containing permissions and return a simple (read, write) pair suitable for use as a callback.

Parameters

- **view** (*str*) – the view to read from, built-in (OE0520) or custom (MOD.TABLE)
- **user_field** (*str*) – the username field in the view. must be a key or browseable field.
- **read_field** (*str*) – the read permission field name - must be bool.

- **write_field** (*str*) – the write permission field name - must be bool.

Returns func

5.4 extools.ui.custom_table

Custom Table UI

A simple UI for any custom table. Simply point this UI class at any table and it will render the fields for you.

If poplar_screenperms is installed, a permission code can be provided to control access to the screen or its fields.

class extools.ui.custom_table.**CustomTableUI**(*tablename*, *permcode*)

Bases: object

CustomTableUI Class

Create a new custom UI for the given table and permissions code.

Parameters

- **tablename** (*str*) – Custom Table name
- **permcode** (*str*) – Permissions code

```
from extools.ui.custom_table import CustomTableUI

TABLENAME = "MYTABLE"
PERMISSION_CODE = "VIMYTA"

def main(*args, **kwargs):
    CustomTableUI(TABLENAME, PERMISSION_CODE)
```

CheckSaveRecord(*cb=None*)

NewHeaderLoaded()

OnDocumentNumberChanged()

UnsavedChangesExist()

UpdateButtons()

UpdateLookupLabel(*fname*)

UpdateLookupLabels()

addButtons()

createScreen()

csr_default(*bProceed*)

ds_onAfter(*e*, *fieldName*, *value*)

ds_onAfterOpen()

ds_onBefore(*e*, *fieldName*, *value*)

fecKeyField_click(*btnType*)

getFieldControlByName(*fname*)

isKeyField(*fieldName*)

loadTable(*tableName*)

name_finder_cancel()

name_finder_ok(*e*)

name_finder_ok_part2(*bProceed*)

onBtnClose_Click()

onBtnClose_Click_part2(*bProceed*)

onBtnDelete_Click()

onBtnDelete_confirmed(*r*)

onBtnSave_Click()

onConfirmDelete(*r*)

onConfirmSave(*r*)

saveRecord()

class extools.ui.custom_table.**EnhancedFieldControl**(*ui*, *fec*)
Bases: object

calcFinderFilter()

click(*btnType*)

finder_cancel()

finder_ok(*e*)

class extools.ui.**ExUI**(*title='exui'*)
Bases: object

An enhanced UI class for extender.

ExUI adds additional helpers to the standard Extender UI class.

BUTTON_SPACE = 150

BUTTON_WIDTH = 1065

FILE_DIALOG_FILTERS = {'excel': ('Microsoft Excel File (*.xls, *.xlsx)|*.xls*|All Fil

FINDER_BUTTON_TYPE = 4

finder_on_click_for(*view*, *ok_callback*, *cancel_callback*, *_filter=''*, *display_fields='1'*, *return_fields='1'*)
Build a callback to execute on finder button click.

Parameters

- **view** (*str*) – The view name (either custom or built in) to find across.
- **ok_callback** (*func*) – Callback to execute on user OK
- **cancel_callback** (*func*) – Callback to execute on user cancel
- **_filter** (*str*) – Filter to apply to finder records.
- **display_fields** (*str (comma separated indexes)*) – fields to display in finder
- **return_fields** (*str (comma separated indexes)*) – fields to display in finder

Returns finder_on_click callback function

get_browse_click_callback (*field*, *title='Select File'*, *_filter=''*)
Create the browse button callback in a closure to pass the field.

get_file_ok_callback (*field*)
Create the File OK callback in a closure to pass the field.

input_with_button (*caption*, *callback*, *default=''*, *label='Button'*)
Create a compound field with an input field and a button:

```
         +-------------------------+  +----------+
caption  | <input field>           |  | <button> |
         +-------------------------+  +----------+
```

Parameters

- **caption** (*str*) – input field caption
- **callback** (*function*) – callback function for the button
- **default** (*str*) – default value for the input field
- **label** (*str*) – label for the button

Returns (input_field, button)

Return type (accpac.UIField, accpac.UIButton)

To create a file browse input and button:

```
file_path_fld, file_browse_btn = self.input_with_button(
        "File", self.on_browse_click, label="Browse")
```

CHAPTER 6

extools.env

A number of helpers are available here for the Extender Python environment. For example: *extools.env.execution_context()* can detect whether a script is running through PS or VI, and whether you can show messages.

Aso included is the `extools.env.StreamConductor`, which allows you to temporarily redirect and capture the standard input and output streams used globally. Useful for getting code to run that expects these streams to be connected.

Tools for the Extender Python environment.

`extools.env.`**`EXEC_PS = 1`**
Process Sceduler Execution Context

`extools.env.`**`EXEC_VI = 0`**
Extender Execution Context

`extools.env.`**`execution_context`**`()`
Are we running through PS or VI?

Returns `EXEC_PS` if PS else `EXEC_VI`

class `extools.env.stream_conductor.`**`StreamConductor`**(*stdout=None*, *stderr=None*, *stdin=None*)
Bases: `object`

The StreamConductor manages standard I/O operations for Extender.

The Extender environment isn't connected to stdout, stderr, or stdin. To allow applications that assume they'll be present to work, StreamConductor patches them to string buffers that can be read from or written to.

Parameters

- **stdout** (*file-like object*) – file-like object for stdout.
- **stderr** (*file-like object*) – file-like object for stderr.
- **stdin** (*file-like object*) – file-like object for stdin.

patch_streams()
Patch stdout, stderr, and stdin.

patched_streams()
Temporarily patch streams, making sure they are restored.

Context manager for working with patched streams temporarily. After the block exits, the streams will be restored to their original state.

```
conductor = StreamConductor()
with conductor.patched_streams():
    print("Hello")

print(conductor.stdout.getValue()) # prints "Hello"
```

unpatch_streams()
Restore streams to their original state.

CHAPTER 7

extools.report

Tools for generating reports.

They are often required, but who ever remembers the params and criteria? These classes are designed to capture that knowledge for re-use.

They should be pretty strightforward to use:

```
report = POPurchaseOrderReport("PO0001", "PO0003")
report.generate(destination="preview")
```

To print to a file:

```
report = POPurchaseOrderReport("PO0001", "PO0003")
report.generate(destination="file", path="C:\Temp\report.pdf")
```

To open the print dialogue, pass the UI instance:

```
report = POPurchaseOrderReport("PO0001", "PO0003", ui=self)
report.generate(destination="file", path="C:\Temp\report.pdf")
```

To override a parameter:

```
report = POPurchaseOrderReport("PO0001", "PO0003", ONHOLD="1")
report.generate(destination="file", path="C:\Temp\report.pdf")
```

To override the selection criteria:

```
criteria = "((POPORH.ORDNUMBER = 'ORD00001'))"

report = POPurchaseOrderReport("PO0001", "PO0003",
        **{"@SELECTION_CRITERIA": criteria})
report.generate(destination="file", path="C:\Temp\report.pdf")
```

class extools.report.**APChequeReport**(***kwargs*)

Bases: *extools.report.ReportWrapper*

parameter_set = 'BKCHKSTK'

parameters = {'APPRUNNUM': '', 'ENDSERIAL': '{to_id}', 'EXTPARAM1': '2', 'EXTPARAM2':

reports = ('APCHK01.RPT',)

class extools.report.**EFTVendorRemittanceReport**(***kwargs*)
Bases: *extools.report.ReportWrapper*

parameter_set = 'ELPAY01'

parameters = {'BATCHTYPE': 'PY', 'DELMETHOD': '1', 'FROMBTCH': 0, 'FROMENTRY': 0, 'TOB'

reports = ('ELPAY04.RPT',)

class extools.report.**InvoiceActionsReport**(***kwargs*)
Bases: *extools.report.ReportWrapper*

parameter_set = 'OEINACTS'

params = {'FROMDATE': '0', 'FROMSHIPMENT': '{from_id}', 'FUNCDECS': '2', 'LEVEL1NAME':

reports = ('OEINACTS.RPT',)

class extools.report.**OEInvoiceReport**(***kwargs*)
Bases: *extools.report.ReportWrapper*

parameter_set = 'OEINV03'

parameters = {'ECENABLED': '0', 'PRINTBOM': '0', 'PRINTED': '1', 'PRINTKIT': '0', 'QTY

reports = ('OEINV03.RPT',)

selection_criteria = '(({{OEINVH.INVNUMBER}} >= "{from_id}") AND ({{OEINVH.INVNUMBER}}

class extools.report.**POPurchaseOrderReport**(***kwargs*)
Bases: *extools.report.ReportWrapper*

parameter_set = 'POPOR04'

parameters = {'ACTIVE': '1', 'BLANKET': '1', 'DELMETHOD': '1', 'DIRECTEC': '0', 'DPRIN'

reports = ('POPOR04.RPT',)

selection_critieria = '\n (({{POPORH1.PONUMBER}} >= "{from_id}")\n AND ({{POPORH1.PONU

class extools.report.**PayrollChequeReportGenerator**(***kwargs*)
Bases: *extools.report.ReportWrapper*

parameter_set = 'BKCHKSTK'

parameters = {'APPRUNNUM': '', 'ENDSERIAL': '{to_id}', 'EXTPARAM1': '2', 'EXTPARAM2':

reports = ('CPCHK4A.RPT',)

class extools.report.**ReportWrapper**(***kwargs*)
Bases: object

generate(*report_name*, *from_id=''*, *to_id=''*, *destination='file'*, *path=''*, *ui=None*)

parameter_set = ''

parameters = {}

reports = []

selection_criteria = ''

wait_for(*path*, *tries=5*, *sleep=3*)

extools.report.**get_report_class_for**(*report_name*)

extools.report.**get_report_class_for_parameter_set**(*paramset*)

extools.report.**get_report_class_for_report_name**(*name*)

CHAPTER 8

Tests

Automated tests have been built for as much of the code as possible.

Code that can run without `accpac` is tested using standard Python unittest.

Code that needs accpac has to be run through the Sage Desktop, or mocked extensively (more to come). Tests are written using the *extools.test.ExTestCase* class to make developement and execution through the desktop easy.

Check out the tests for the project to get an understanding of how to test code for Extender or to get a feel for the extools internals.

8.1 extools tests

class extools.tests.test_extools.**TestExTools**(*methodName='runTest'*)
Bases: unittest.case.TestCase

class extools.tests.extest_extools.**ExToolsTestCase**(*log_level=15*)
Bases: *extools.test.ExTestCase*

Test the functions in *extools* module.

test_lines_in()
Verify that all_lines_in iterates in the right order and the correct number of times.

test_lines_in_view()
Verify that all_lines_in_view iterates in the right order and the correct number of times.

test_open_and_fetch_row()
Verify that open_and_fetch_row opens and seeks correctly.

extools.tests.extest_extools.**main**(**args*, ***kwargs*)
This main hook is picked up by ExTestRunner for automatic execution.

8.2 exview tests

ExTests for *extools.view.ExView*.

class extools.view.tests.extest_exview.**ExViewAttributesTestCase**(*log_level=15*)
Bases: *extools.test.ExTestCase*

Verify *extools.view.ExView* passes attrs to the view.

setup()
Clear the class internals before each test.

setup_class()
Generate the attribute proxy tests.

class extools.view.tests.extest_exview.**ExViewCMTestCase**(*log_level=15*)
Bases: *extools.test.ExTestCase*

Test the *extools.view.exview* context manager.

test_exview_contextmanager()
Verify that the *extools.view.exview()* context manager is working:

- Opens and seeks to the first record by default.
- Opens and seeks to the provided criteria.
- Raises on an invalid view id.

class extools.view.tests.extest_exview.**ExViewExceptionsTestCase**(*log_level=15*)
Bases: *extools.test.ExTestCase*

Verify *extools.view.ExView* raises on non-zero returns.

generate_raise_tests()
Generate mocked raise tests for all wrapped methods.

Dynamically create a test for each method in *extools.view.ExView.WRAP*. This should be run during the `.setup_class()` hook so that the tests are defined before `.run()` is called.

setup_class()
Dynamically generate the tests for wrapped methods.

class extools.view.tests.extest_exview.**ExViewInternalsTestCase**(*log_level=15*)
Bases: *extools.test.ExTestCase*

Verify *extools.view.ExView* internals are working.

setup()
Clear the class internals before each test.

test_compose_optfield_view_detection()
Verify that optfield views are detected and helpers added.

test_composed_view_list()
Verify that the list of composed views is determined correctly.

test_detail_view_detection()
Verify that detail views are detected and helpers added.

test_detail_view_wrapper()
Verify that the detail view wrappers work as expected.

test_field_names()
Verify that the list of fiel names is determined correctly.

test_index_detection()
Verify indexes are identified through introspection on init.

test_intial_optfield_view_detection()
Verify that optfield views are detected and helpers added.

test_optfield_view_wrapper()
Verify that the optfield view wrappers work as expected.

test_view_cache()
Verify that the view cache is populated correctly.

class extools.view.tests.extest_exview.**ExViewProxyMethodsTestCase**(*log_level=15*)
Bases: *extools.test.ExTestCase*

Verify *extools.view.ExView* proxy methods are working.

setup()
Clear the class internals before each test.

test_all()
Verify that `all` yields all records in a view.

test_create()
Verify that `where` yields all matching in a view.

test_where()
Verify that `where` yields all matching in a view.

class extools.view.tests.extest_exview.**ExViewViewCacheTestCase**(*log_level=15*)
Bases: *extools.test.ExTestCase*

Verify *extools.view.ExView* view cache is working.

setup()
Clear the class internals before each test.

test__view_cache()
Verify that _view_cache returns the cache namespace.

test_cached_view()
Verify that cached_view returns views in the cache namespace.

test_cached_view_create()
Verify that cached_view returns view in the cache namespace.

test_remove_cached_view()
Verify that remove_cached_view removes from the cache namespace.

extools.view.tests.extest_exview.**main**(**args*, ***kwargs*)
This main hook is picked up by ExTestRunner for automatic execution.

CHAPTER 9

Notes

9.1 Custom Table Fields Reference

Custom Tables support a number of different field types.

An example module containing all the types:

```
[MODULE]
id=TEST
name=Test
desc=Test
company=Test

[TABLE]
name=TEST.TEST
desc=Test table to get a handle on field types.
dbname=TEST

[FIELD1]
field=TEXT
datatype=1
mask=%-60C
size=60

[FIELD2]
field=DATE
datatype=3

[FIELD3]
field=TIME
datatype=4

[FIELD4]
field=INT16
```

(continues on next page)

(continued from previous page)

```
datatype=7

[FIELD5]
field=INT32
datatype=8

[FIELD6]
field=BOOL
datatype=9

[FIELD7]
field=NUMBER #aka BCD
datatype=6
size=10
decimals=0

[KEY1]
KEYNAME=KEY1
FIELD1=TEXT
allowdup=0
```

Some collected notes on working with Extender. First up: field types. Easily determined by building a table in the Custom Table tool and exporting the module.

extools is created and maintained by Poplar Development. The full source is available on Bitbucket.

Before we get started, what brings you here today? Are you looking to...

- Work with the built-in Extender views but write more idiomatic code? Check out *extools: foundation*.
- Write Extender scripts using `try/except` instead of `if r != 0`, views that compose themselves, automatic optional field helpers, intelligent seek, and more? Try *extools.view* on for size.
- Handle messages and logging easily and consistently in your Extender scripts? Have a look at *extools.message*.
- Automate the testing of your code? The *extools.test* package may be just what you need.
- Get a better handle on the environment in which your scripts are running? is this being called from Process Scheduler, an import, or a view? Helpers in *extools.env* may help.
- Write scripts that need to be run from the panel but don't need a UI? *extools.ui.bare* helps with that.
- Add a new column to pretty much any grid in just a few lines? *extools.ui.callback_column.CallbackColumnUI*, *extools.ui.datasource_column.DatasourceColumnUI*, *extools.ui.optfield_column.OptfieldColumnUI* have you covered.

CHAPTER 10

extools: foundation

Full of little helpers that can be used to make Extender scripts more idiomatic, these functions are the foundation for the other tools in the package. Things like checking the success of view operations in bulk, or generators that yield all records in a view. Instead of

```
_rc = view.recordClear()
_br = view.browse("")

if _rc != 0 or _br != 0:
    showMessage("Custom Script: failed to clear or browse.")
    return False

while view.fetch() == 0:
    val = view.get("FIELD")
    # do stuff
```

Use the helpers in extools to make things more idiomatic.

```
from extools import (success, all_records_in, )

_rc = view.recordClear()
_br = view.browse("")

# Avoid numeric comparisons that can be hard to understand.
if not success(_rc, _br):
    # Generate messages
    alert("failed to clear or browse.")
    return False

while success(view.fetch()):
    val = view.get("FIELD")
    # do stuff

# alternatively, use a for loop and the all_records_in helper.
```

(continues on next page)

(continued from previous page)

```
for item in all_records_in(view):
    val = item.get("FIELD")
```

CHAPTER 11

extools.messages: a simple logging framework for Extender scripts

Messages provides the ExMessages class, which acts like a logger but shows messages on screen (as well as writing to a file if required). It provides configurable log levels, so the verbosity of the module is easily controlled.

Set it to DEBUG while developing and then to WARNING before releasing, be confident that you haven't left a hanging debug message lying around.

Instead of having to uncomment those hidden showMessages when trying to fix a problem in place, just change the log level and undo the Extender script check-in when you've finished troubleshooting.

And keep DRY. Set the log level once, and the message output format, along with titles, once. The user experience will be more consistent and the code must easier to maintain.

```
from extools.message import ExMessages
from extools.env import vidir_path

exn = ExMessages("My Customization",
                 level=ExMessages.INFO,
                 logpath=vidir_path / "mycust.log")

# Write to the log and display a message box with "My Customization" as a
# header and the message as content.
exn.info("This is an info message.")

# Would display a message box with "DEBUG - My Customization" as a header
# and the message as content but suppressed due to level.
exn.debug("This is a debug message.")

# Update the log level on the fly.
exn.level = ExMessages.DEBUG

# This time it will work.
exn.debug("This is a debug message.")
```

As an added bonus messages logged at the panic, critical, and error level can include traceback information - allowing the capture of deep tracebacks that exceed the Sage system message size.

```
# Append the traceback for the last exception to the
# message.
exn.error("This error occured!", exc_info=True)
```

CHAPTER 12

extools.view: a wrapper around Extender views that raises

ExView is an exception raising wrapper around the standard Extender View object. It has some other extensions as well, such as built in generators, that make working with with views more pythonic.

```
from extools.message import ExMessages
from extools.view import ExView, ExViewError

exm = ExMessages("MYMOD")

try:
    # Open the AR Items view
    exv = ExView("AR0010")
    # Compose, adding the ar0009 (aritd) and ar0011 (aritt) views
    exv.compose()
    for item in exv.lines():
        for price in item.ar0009.lines():
            # Do some stuff with the item prices
except ExViewError as e:
    # Use the descriptive message in the exception.
    exm.error("Failed to update pricing, {}.".format(e))
```

For more information on ExView's self-composing feature, see the doc on *Self-composing views*.

CHAPTER 13

extools.env: details on the current execution env

Sometimes you need details about the execution environment from within a customization. What is the VI root directory? Where can I put a temp file? Is this script executing from Process Scheduler?

The environment package leverages Python's Path library to make working with the environment easy.

Python Module Index

e

Index

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y