Author: Patrick Emery, Matrix Science Ltd
Version: 2024-03-13

Creating a custom Distiller report

Reports in Mascot Distiller 2.8 and later are generated using Python scripts and our Mascot Parser API. A report consists of two separate files:

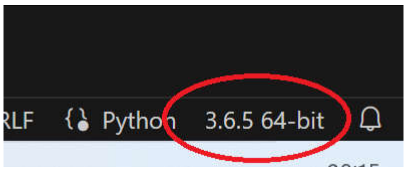
1. The main report Python file
2. An XML file which defines the report inputs, and any GUI wizard pages required to take input options from the user.

In this tutorial, we'll take a look at how to create your own custom reports for Mascot Distiller. In order to write your own reports, you should have a good working knowledge of the Python programming language, and also be familiar with our Mascot Parser library, used to access the search and quantitation results (https://www.matrixscience.com/msparser.html).

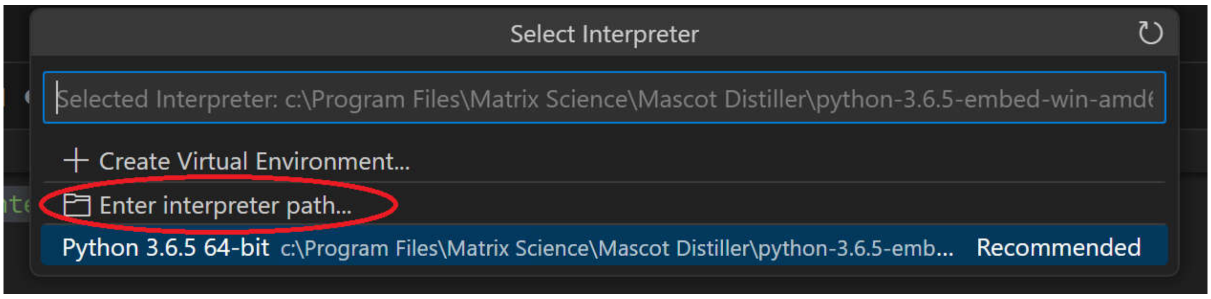
Setting up your development environment

Mascot Distiller ships with an embedded version of Python 3.6 and a number of useful libraries including Mascot Parser, pandas, Matplotlib and others. In order to develop your own reports, you want to use a matching environment. If you're running development on the same PC as Distiller is installed on, you can do this by simply pointing your development environment to use the Python installed by Distiller (C:\Program Files\Matrix Science\Mascot Distiller\python-3.6.5-embed-win-amd64 by default). For example, in Microsoft's free Visual Studio Code:

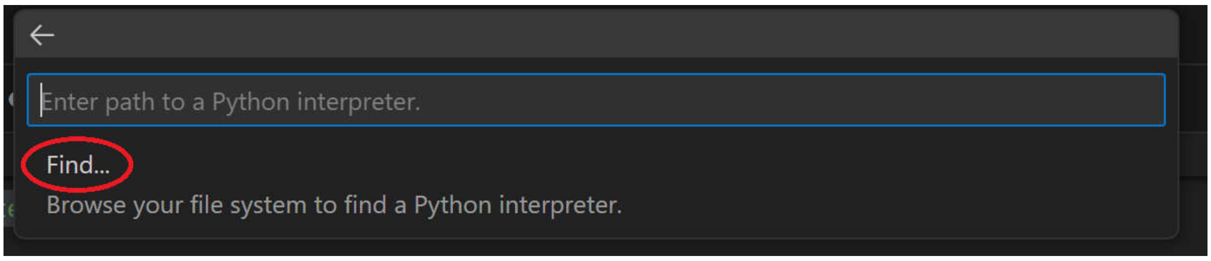
Click on the Python version number in the bottom right of the editor



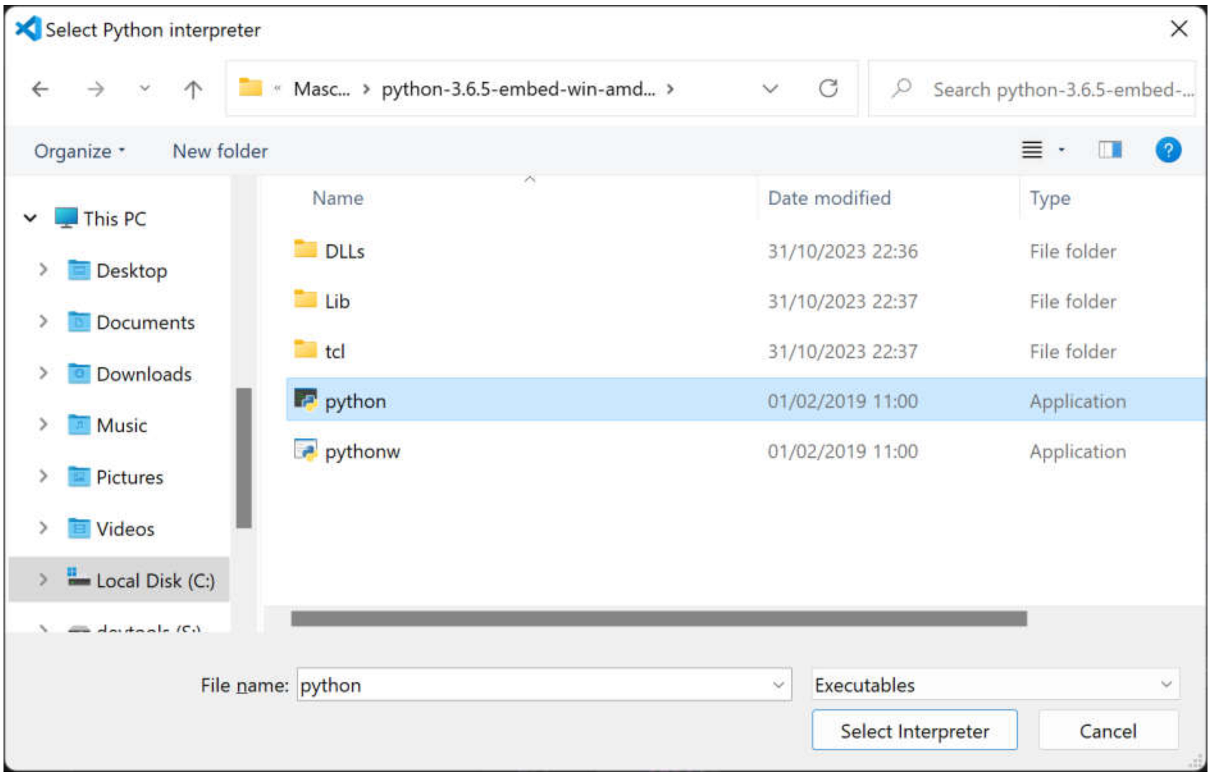
This will open a 'dialog' in the top middle of the editor pane. Click on "Enter interpreter path...":


Select Interpreter
Selected Interpreter: c:\Program Files\Matrix Science\Mascot Distiller\python-3.6.5-embed-win-amd
Create Virtual Environment...
Enter interpreter path...
Python 3.6.5 64-bit c:\Program Files\Matrix Science\Mascot Distiller\python-3.6.5-emb... Recommended

This will change the interface – now click on "Find":


Enter path to a Python interpreter.
Find...
Browse your file system to find a Python interpreter.

And browse to the Python executable located in “C:\Program Files\Matrix Science\Mascot Distiller\python-3.6.5-embed-win-amd64”:



Name	Date modified	Type
DLLs	31/10/2023 22:36	File folder
Lib	31/10/2023 22:37	File folder
tcl	31/10/2023 22:37	File folder
python	01/02/2019 11:00	Application
pythonw	01/02/2019 11:00	Application

File name: python

If you’re developing the report on a different PC, you can install Mascot Distiller for free on your development PC. This will install Distiller in viewer mode, which includes the embedded Python and is sufficient for developing and testing your own Python reports.

Example report – Top 3 Intensity

As an example of how to create a custom report for Mascot Distiller, we’ll work through creating a new report which calculates the Average (top-3) protein intensity for all sample components and generates a CSV export file.

XML configuration file

As mentioned above, a Distiller report requires two separate files – the Python source file which runs and generates the report, and an XML file which defines the report inputs and outputs, the supported quantitation protocols and the output file type. The report file can be called anything of your choosing, although the filename should end with the extension “.py”. The XML file must match the name of your Python report file, with “.xml” appended to the end. So we’ll call our top-3 intensity report files:

- top-3.py
- top-3.py.xml

We’ll start by defining the XML file. This should implement the “distiller_report_definition_1.xsd” XML schema, which is shipped with Mascot Distiller. You’ll find a copy of it in the Mascot Distiller installation directory. The XML file defines which quantitation protocols the report supports, so that Mascot Distiller will disable it if you don’t have search results of the correct type, and where in the menu the report will appear in Distiller, as well as any inputs the report requires.

DistillerReport is the root element. The “title” attribute should be set to give the name of the report, and the grouping attribute the “path” to the report in the GUI. In our example, these are set to “Top 3 protein intensity” and “Custom” respectively:

```
<DistillerReport majorVersion="1" minorVersion="0" title="Top 3 protein
intensity" grouping="Custom"
```

Our report would be accessible in the Distiller GUI under Analysis->Reports->Custom->Top 3 protein intensity. The grouping attribute can describe a more complex path – for example Custom/Intensity.

The Supports element has a series of attributes which can be set to true or false to define which quantitation protocols (https://www.matrixscience.com/help/quant_overview_help.html) the report supports. In this case, we’ll write the report to use data from any of the quantitation protocols supported by Mascot:

```
<Supports average="true" precursor="true" replicate="true" reporter="true"
multiplex="true"/>
```

The next section of the XML file defines the report inputs. There are two sections to this. The “ReportConfiguration” element defines fixed input values which are passed through to the report and also available to Distiller. For our example, we want to tell Mascot Distiller that the report will generate a CSV file:

```
<Parameter name="exportFormat" label="Format" value="CSV" type="text"
mapsTo="ExportFileType"/>
```

This maps to a standard report variable – “ExportFileType”.

We also want to define a variable that can be used later on in the XML to decide whether or not to show a Wizard page to the user to select a contaminants database for exclusion from the report:

```
<Parameter name="databaseCount" label="no databases" type="integer"
value="@{DatabaseNames.Count}"/>
```

The @{DatabaseNames.Count} value tells Distiller to substitute in the number of databases, while the type tells it that this is an integer value.

The “Wizard” element defines any GUI Wizard which should be displayed to the user running the report in order to set any required values. After opening the Wizard element, you need to define a “WelcomeText” element – this creates an opening page for the Wizard describing the report. After this we add one or more “Page” elements, which should define inputs for the user to make. Finally, we end with a “CompletionText” element, defining a final page to display before the user runs the report.

For example, the XML below adds a Page which defines a drop-down list from which the user can choose the peptide selection criteria for the Average top-3 protein component intensity calculation:

```xml
            <!--Subsequent pages which take user inputs are defined by Page
elements.  The value of the "title" attribute will be displayed-->
            <!--at the top of the page in the GUI-->
            <Page title="Peptide selection criteria">
                <!--HelpText is displayed above the page parameters to give
instructions to the user-->
                <HelpText>The selection type determines whether the n peptides
must have different sequences (unique_sequence) or whether to accept different
modification states of same sequence (unique_mr), or even to accept peptides
with same sequence and modifications in different charge states
(unique_mz)</HelpText>
                <!--On the first page, we'll define the peptide selection
criteria.  This is a drop down "select" box with the 3 supported-->
                <!--options, allowing the user to pick one.  You can find a
description of the options at-->
                <!--http://www.matrixscience.com/help/quant_average_help.html-
->
                <!--The selected option is set to the parameter
"selectionType", which can then be accessed in the Python script-->
                <Parameter name="selectionType" label="Selection type"
type="select">
                    <!--Set the default selected option using the selected
attribute-->
                    <Option value="unique_sequence" displayString="Unique
sequence" selected="true"/>
                    <!--"value" is the value which will be set to
selectionType if the option is selected-->
                    <Option value="unique_mr" displayString="Unique Mr"/>
                    <!--and displayString defines the text which will be
displayed for the option in the GUI-->
                    <Option value="unique_mz" displayString="Unique M/Z"/>
                </Parameter>
            </Page>
```

Python report file

The Python environment supplied with Mascot Distiller includes a number of useful libraries, such as our Mascot Parser library for parsing Mascot search and quantitation results and a number of third party libraries including pandas. Additionally, we ship a number of helper libraries which simplify extracting the required result values, which we'll use to simplify the initial loading of the search and quantitation results.

At the head of the Python script, we'll import the libraries we want to use:

```
import sys
import CreateQuantDataFrames
import pandas
import msparser
import io
import WriteReports
import LoadQuantitation
```

msparser in the library name for Mascot Parser. CreateQuantDataFrames, WriteReports and LoadQuantitation are all helper libraries that we'll be using to load the data into pandas dataframes, and to format the final report. These transform the results into pandas (https://pandas.pydata.org/) data structures. You can find these in the "reports" directory in the Distiller Python installation. Additionally, we'll be using msparser to access the results.

sys and io are standard Python libraries.

In the body of the script, the first thing we want to do is initialise a global msparser logging object. This will output anything we choose to log to the standard output of the script – Distiller will capture this and include it in its own log file (provided the required log level is set). You can find more information on logging in the Mascot Parser API documentation.

```
# create an msparser logger
mylogger_ = msparser.ms_stdout_logger()
```

Now we'll start the actual main method of the report. The report script itself is heavily commented to explain what the various steps and calls are doing, so we won't run through every line here. If you have any questions regarding the script, please contact Matrix Science support.

Mascot Distiller passes the report settings out to the Python script as a CSV file. If you want to take a look at the contents of the file being passed to a report, you can do so by setting the Distiller debug levels. Under Tools->Log->Preferences, include "Debug 1", and the contents of the CSV file will be added to the Distiller log file when you run a report. To simplify parsing the CSV file to extract the required settings, we'll load the CSV into a PANDAS dataframe:

```
    propsPath = sys.argv[1]
    # Load in properties.csv - this is the settings file written out to Python 
from Distiller
    # with the report configuration to run with
    props_csv = pandas.read_csv(propsPath, delimiter = ',', header = 0, 
keep_default_na = False, quotechar="\"")
```

And then make subsets of the settings to more easily access the values – for example, we'll get the "selectionType" value we defined in the .xml file and which the user has set in the Wizard and set it to a variable called `peptideSelectionType`:

```
    # Create subsets of the properties file
    props_logging = props_csv[props_csv.Identifier == ‘LoggingOptions’] # Log
and column mask
    props_summary = props_csv[props_csv.Identifier == ‘SummaryInputs’] #
creating peptide summary flag settings
    props_selectionType = props_csv[props_csv.Identifier == ‘selectionType’] #
Peptide selection criteria for the top-3 calculation.  See top-3.py.xml
    peptideSelectionType = props_selectionType.Input1.iloc[0]
    excludeDatabase = float(props_analysis.Input5.values[0]) if
props_analysis.Input5.values[0] != ‘’ else ‘’ #(numeric) Database to exclude
in analysis.  If 0, all databases are used
    savePath = props_options.Input2.iloc[0]    # file path to save the report
to.
```

The next part of the method calls a helper method in LoadQuantitation to load our search and quantitation results using the relevant settings we’ve just loaded in from the csv properties file:

```
    # load the peptide summary, quantitation results etc using the helper
method in the supplied LoadQuantitation file
    loadRes = LoadQuantitation.DoLoad(propsPath)
```

This returns an array which we’re setting to a variable called loadRes, which we’ll pass to a new method in our script, along with the other required settings, which will generate the actual report:

```
    # call the function to generate the report
    CreateTop3Report(loadRes, excludeDatabase, peptideSelectionType, savePath,
props_header, props_rawfiles, exportHeader)
```

We’ve documented the input parameters for the function in the script – notice that we’re passing through our loaded results array, details of any contaminants database, the peptide selection type selected by the user etc.

```
## Creates the top-3 intensity report
# \param loadedResults The loaded search and quantitation results.
# \param excludeDatabase The contaminants database index to exclude from the
report (if set)
# \param peptideSelectionType The peptide selection/grouping criteria selected
by the user on the report Wizard
# \param savePath The report export path defined by Distiller
# \param props_header Header properties passed in from Distiller
# \param props_rawfiles Raw file properties information passed in from
Distiller
# \param exportHeader String set to "True" or "False" depending on the user
selection on the report Wizard
def CreateTop3Report(loadedResults, excludeDatabase, peptideSelectionType,
savePath, props_header, props_rawfiles, exportHeader):
    # Check we have results
    if (len(loadedResults) == 0):
        sys.exit("No Results Parameters Supplied")
```

The first step in the method is to sanity check that we have loaded search and quantitation results – if not, exit the script with an error message.

Next, we need to access the search and quantitation results and determine which quantitation protocol was used (reporter, precursor, label free, average etc), as the way we need to access the actual peptide component intensity values will depend on this – for example, to check if we have “Average (top-3)” label free quantitation results, check if the quantitation method protocol has the Average method set:

```
    # Get the msparser ms_peptidesummary, quantitation results back from the
loadedResults array
    isMS1 = loadedResults[0].isMS1 # MS1 quantitation results?  True or false
    quant = loadedResults[0].qObj # The quantitation results (either
ms_ms1quantitation or ms_ms2quantitation depending on whether or not isMS1 is
true)
    pepSum = loadedResults[0].pepSum # The ms_peptidesummary (protein hits)
    qMethod = loadedResults[0].qMethod # The quantitation method definition
(ms_quant_method)

    # MS2, Precursor and Average quantitation methods all require different
handling to find the individual peptide component intensities

    isAverage = False
    if isMS1:
        isAverage = qMethod.getProtocol().getAverage() != None
```

Then we need to get the individual component (sample) names from the quantitation method:

```
    # Get the component names
    componentNames = []
    if isAverage:
        componentNames.append('Avg')
    else:
        for c in range(0,qMethod.getNumberOfComponents()):
            component = qMethod.getComponentByNumber(c).getName()
            componentNames.append(component)
```

We’ll be calculating the top-3 intensity protein intensity values for each component/sample individually. We’ll need to get our protein hits from msparser for this, using one of our supplied helper methods to pull all the anchor proteins into an array, which we’ll call proteins:

```
    # Helper method which pulls the anchor protein ms_protein results from the
peptide summary into an array.
    proteins = CreateQuantDataFrames.pullProteinsFrom(pepSum)
```

Now we can loop through each protein and calculate the sample intensity values:

```
    end = len(proteins)
    # this will be an array of arrays containing the data to export to the CSV
file
    data = []
    for i in range(0, end):
        # output progress information
        WriteReports.OutputProgress('Calculating protein intensity values',
(i+1), end)
```

WriteReports.OutputProgress allows us to pass progress information back to Mascot Distiller for display to the user. The text will be displayed in the progress bar, while (i+1) and end are out current position in the loop and the final point respectively – these will be converted to a percentage completed.

Now, we access the current protein at the index i in our proteins array and set up a new row for the report table, skipping any protein that comes from the user specified contaminants database (if any):

```
        row = []
        protein = proteins[i]

        # skip any protein hits from the selected contaminants database
        if protein.getDB() == excludeDatabase:
            continue
```

We'll start our output row with the protein hit number and accession:

```
        hitNo = protein.getHitNumber()
        if protein.getMemberNumber() > 0:
            row.append("{}.{}".format(hitNo,protein.getMemberNumber()))
        else:
            row.append(hitNo)
        row.append(protein.getAccession())
```

Now we can calculate the protein intensity for each component. We loop through the componentNames array we defined earlier, and call another procedure defined later in the script – calculateProteinIntensity – passing in the ms_protein instance along with the ms_peptidesummary, quantitation results, component name, method details and the user defined peptide selection criteria.

This returns the calculated average intensity for the protein sample, and we'll add it to the row:

```
       for name in enumerate(componentNames):
            intensity = calculateProteinIntensity(protein, pepSum, quant,
name, isAverage, isMS1, peptideSelectionType)
            if intensity > 0:
                row.append(intensity)
            else:
                row.append('')
```

calculateProteinIntensity is fully documented in the script source code, so you can see there how the average intensity values are calculated. We'll end the protein row with the description taken from the FASTA entry, and then append the row into our data array:

```
        row.append(pepSum.getProteinDescription(protein.getAccession()))
       data.append(row)
```

This procedure is repeated for each protein hit in the results. Once that is completed, we need to put together the column header information:

```
    # create a pandas DataFrame from our data array with column headers
    header = []
    header.append(‘’)
    header.append(‘Accession’)
    for name in componentNames:
        header.append(“{} {}”.format(name,’intensity’))
    header.append(‘Description’)
```

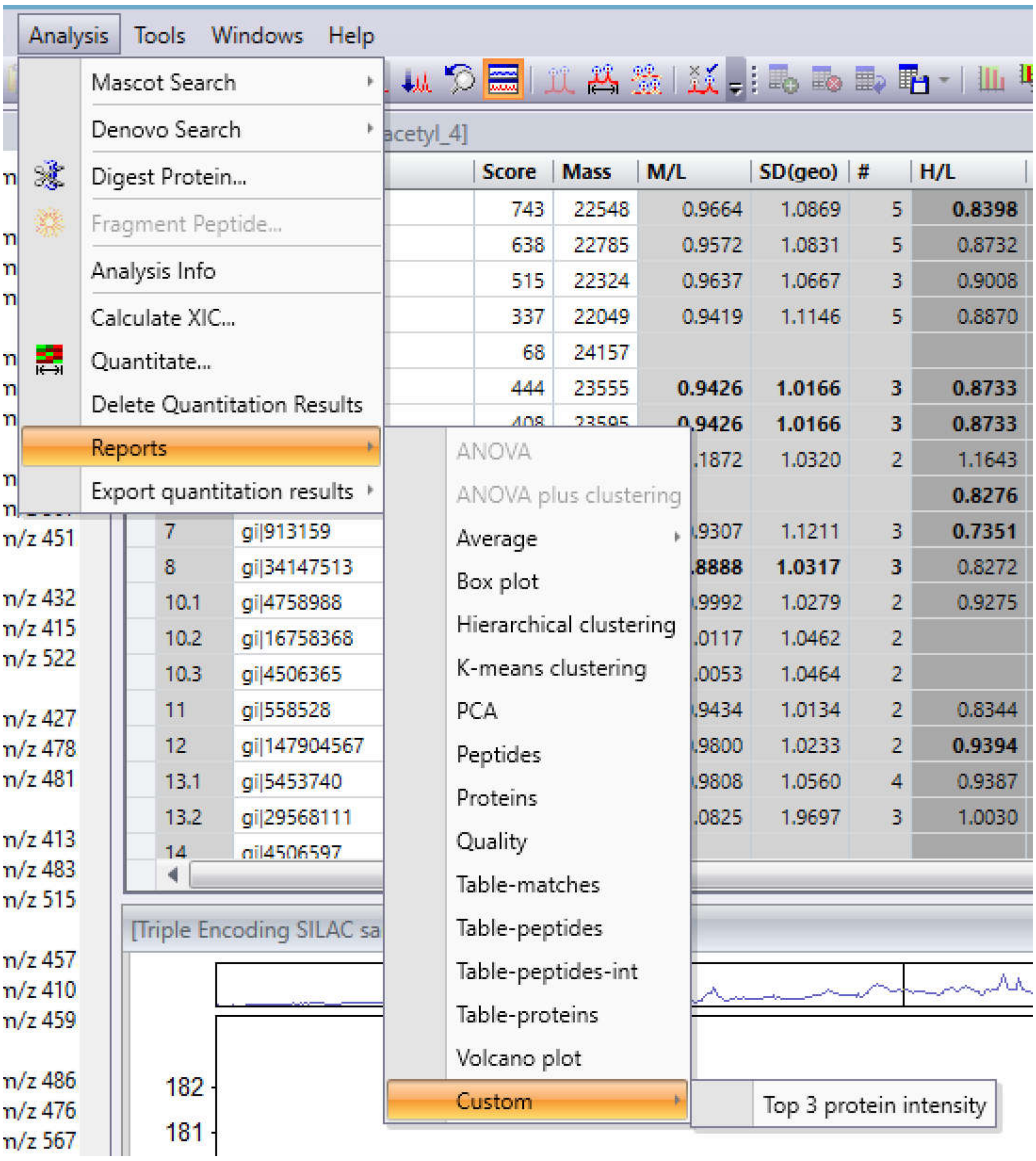
We can then convert our data array containing all the protein intensity information into a pandas data frame with our header:

```
    # pull the data into a Pandas DataFrame
    # to easily output a CSV file with a header but no row index
    df = pandas.DataFrame(data, None, header)
```

The advantage of doing this is that the pandas library has methods which will allow us to easily export the data into a CSV file without having to write any of our own export code, simply by calling the to_csv method of the data frame:

```
    if exportHeader == 'True':
# user has requested the complete report header be exported
        # code snipped, see the Python script
    else:
        # if we don't want to add a header, can write directly to the file
path rather than a string:
        df.to_csv(savePath, ',','',None,None,True,False)
```

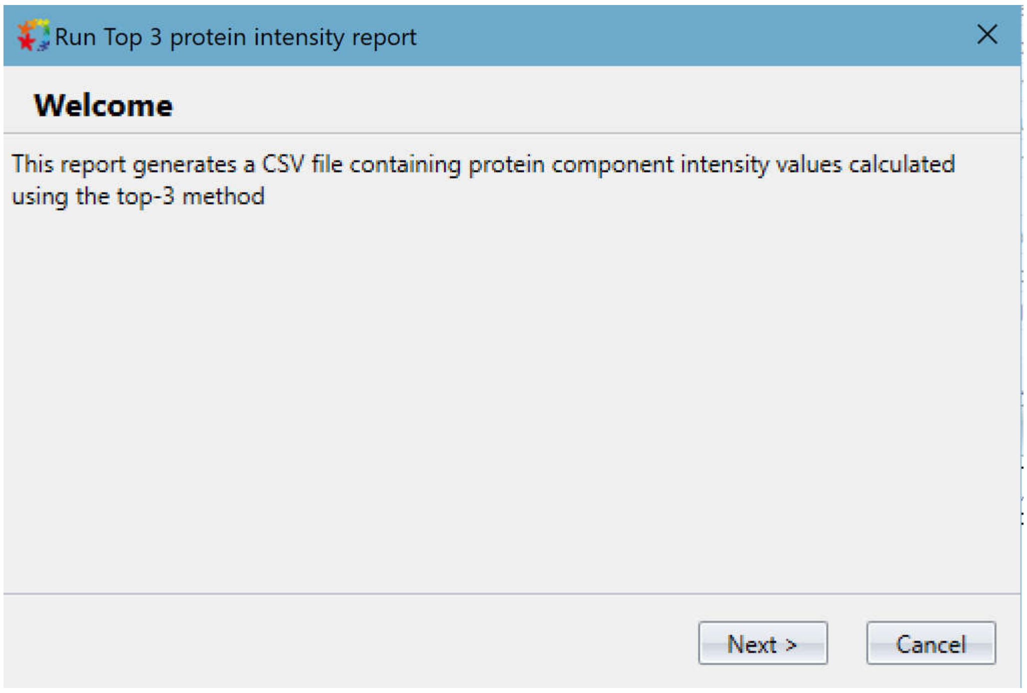
Our report script is now ready. Copy the .xml and .py files to “C:\ProgramData\Matrix Science\Mascot Distiller\reports” directory on the workstation with your Mascot Distiller information. Restart Mascot Distiller and you'll see your custom report appearing in the Mascot Distiller in the path defined in the XML file.



		Score	Mass	M/L	SD(geo)	#	H/L
		743	22548	0.9664	1.0869	5	**0.8398**
		638	22785	0.9572	1.0831	5	0.8732
		515	22324	0.9637	1.0667	3	0.9008
		337	22049	0.9419	1.1146	5	0.8870
		68	24157				
		444	23555	**0.9426**	**1.0166**	3	**0.8733**
		408	23595	**0.9426**	**1.0166**	3	**0.8733**
				.1872	1.0320	2	1.1643
							0.8276
7	gi|913159			.9307	1.1211	3	**0.7351**
8	gi|34147513			**.8888**	**1.0317**	3	0.8272
10.1	gi|4758988			.9992	1.0279	2	0.9275
10.2	gi|16758368			.0117	1.0462	2	
10.3	gi|4506365			.0053	1.0464	2	
11	gi|558528			.9434	1.0134	2	0.8344
12	gi|147904567			.9800	1.0233	2	**0.9394**
13.1	gi|5453740			.9808	1.0560	4	0.9387
13.2	gi|29568111			.0825	1.9697	3	1.0030
14	gi|4506597						

***Note:** Once the script is registered, you can update the .py or .xml files without restarting Mascot Distiller. The only change that won't be reflected without restarting Distiller are any changes made to the "grouping" element of the XML file.*

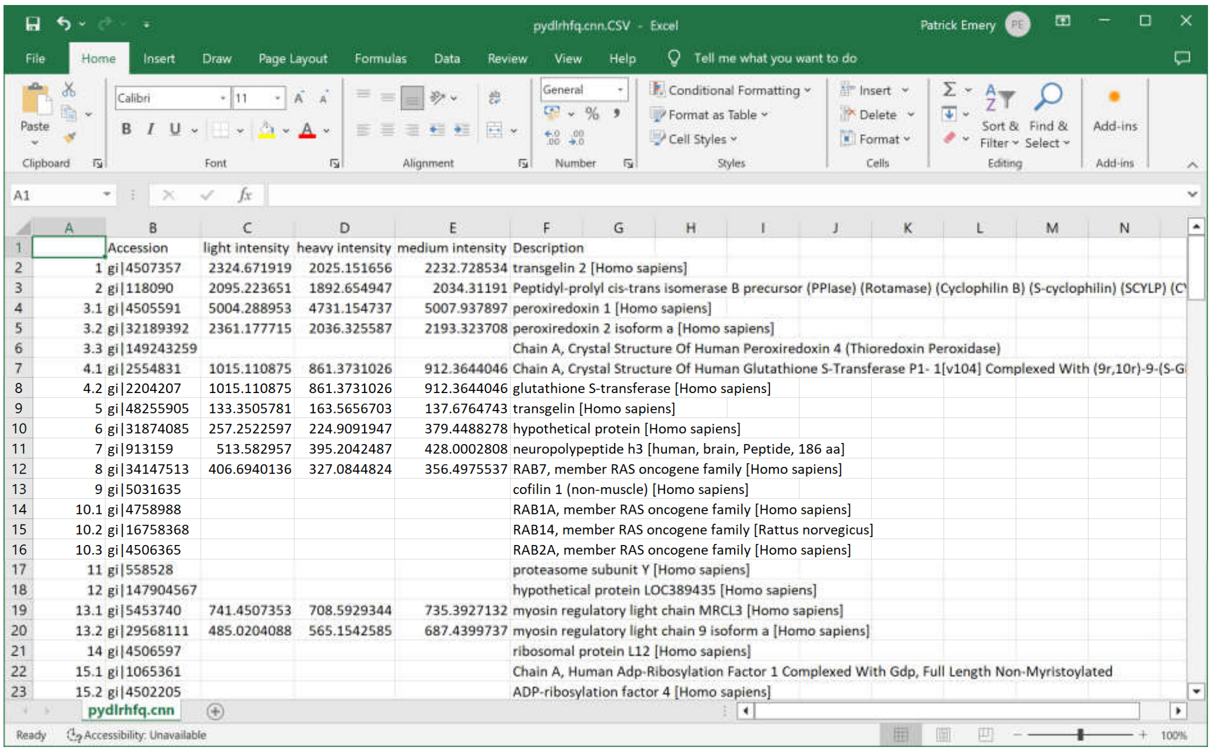
If we now click to run the report, Mascot Distiller will generate the report wizard automatically from the XML, allowing the user to run the report:


Run Top 3 protein intensity report

Welcome

This report generates a CSV file containing protein component intensity values calculated using the top-3 method

When completed, the generated CSV file will open automatically in the software registered on your system to handle CSV files – in our case, Excel:



	Accession	light intensity	heavy intensity	medium intensity	Description
1	gi|4507357	2324.671919	2025.151656	2232.728534	transgelin 2 [Homo sapiens]
2	gi|118090	2095.223651	1892.654947	2034.31191	Peptidyl-prolyl cis-trans isomerase B precursor (PPIase) (Rotamase) (Cyclophilin B) (S-cyclophilin) (SCYLP) (C
3.1	gi|4505591	5004.288953	4731.154737	5007.937897	peroxiredoxin 1 [Homo sapiens]
3.2	gi|32189392	2361.177715	2036.325587	2193.323708	peroxiredoxin 2 isoform a [Homo sapiens]
3.3	gi|149243259				Chain A, Crystal Structure Of Human Peroxiredoxin 4 (Thioredoxin Peroxidase)
4.1	gi|2554831	1015.110875	861.3731026	912.3644046	Chain A, Crystal Structure Of Human Glutathione S-Transferase P1- 1[v104] Complexed With (9r,10r)-9-(S-G
4.2	gi|2204207	1015.110875	861.3731026	912.3644046	glutathione S-transferase [Homo sapiens]
5	gi|48255905	133.3505781	163.5656703	137.6764743	transgelin [Homo sapiens]
6	gi|31874085	257.2522597	224.9091947	379.4488278	hypothetical protein [Homo sapiens]
7	gi|913159	513.582957	395.2042487	428.0002808	neuropolypeptide h3 [human, brain, Peptide, 186 aa]
8	gi|34147513	406.6940136	327.0844824	356.4975537	RAB7, member RAS oncogene family [Homo sapiens]
9	gi|5031635				cofilin 1 (non-muscle) [Homo sapiens]
10.1	gi|4758988				RAB1A, member RAS oncogene family [Homo sapiens]
10.2	gi|16758368				RAB14, member RAS oncogene family [Rattus norvegicus]
10.3	gi|4506365				RAB2A, member RAS oncogene family [Homo sapiens]
11	gi|558528				proteasome subunit Y [Homo sapiens]
12	gi|147904567				hypothetical protein LOC389435 [Homo sapiens]
13.1	gi|5453740	741.4507353	708.5929344	735.3927132	myosin regulatory light chain MRCL3 [Homo sapiens]
13.2	gi|29568111	485.0204088	565.1542585	687.4399737	myosin regulatory light chain 9 isoform a [Homo sapiens]
14	gi|4506597				ribosomal protein L12 [Homo sapiens]
15.1	gi|1065361				Chain A, Human Adp-Ribosylation Factor 1 Complexed With Gdp, Full Length Non-Myristoylated
15.2	gi|4502205				ADP-ribosylation factor 4 [Homo sapiens]