Code Standards/Style Guide

Potential references:
http://perldoc.perl.org/perlstyle.html
http://legacy.python.org/dev/peps/pep-0008/

The main focus should be on readability and consistency. Lines should be kept under 110 characters when possible, though this is not a hard rule but a guideline.

Whitespace

0 or 1 lines of whitespace between lines of code, except for closing braces of nested blocks, which should not have lines of whitespace between them. Parameter lists should have exactly one line of space between them and the beginning of code logic. For example, an if block followed by a while loop:

```
if ($a == 1) {
   <code>
   <code>
} # end if

while ($a < 15) {
   <code>
   <code>
}
```

A second example, and if block nested within a while loop:

```
while ($a < 15) {
   <code>
   <code>
   if ($a == 1) {
      <code>
      <code>
   } # end if
}
```

A third example, a subroutine with parameters followed by code logic:

```
sub example {
   my %args = @_;
   $args{p1} = $args{p1} // $default;

   <code>
   <code>
}
```

1 space before/after curly braces, relational operators.

Tab Indentation

Tab = 4 Spaces
No hard tabs (should be converted to whitespace)

If/Else Formatting

elsif and else statements should start on their own line following their related if
Ed Note: No spaces after (or before), a space needed between) and {

```
if ($a == 1) {
    <code>
}
else {
   <code>
}
```

Assignment Formatting

When assigning values to a hash, a single space should separate the keys from the => operators. The => operators should not be aligned.

```
$rv = {
      rv => ...,
      err_type => ...,
      err_msg => ...,
      err_cls => ...
};
```

Hash Key Formatting

When assigning values to a hash, the keys should be written without quotation marks whenever possible. For example

```
$hash_variable = {
      key_one => ..., # This is the preferred format
      'key_two' => ..., # This should be avoided
};
```

SQL Statement Formatting

Queries should be enclosed in qw{} instead of regular quotes and should follow indentation rules similar to the example below. This includes 4-space indentations and capitalization of all SQL keywords.

When using an sql alias multiple times, make sure that the alias uses <alias>_<description> to distinguish between the two. Example: portal_mh_account pma could be included twice in a query as pma_broker and pma_parent.

All table names in 'FROM' and 'JOIN' statements must be followed by 'WITH (NOLOCK)'.

```
SELECT
    op.city AS prop_city
  , ol.portal_eligible_bool AS eligible
  , CASE
        WHEN ol.fk_supplier_id IS NULL THEN 'AgentAdvantage'
        WHEN op.mls_desc IS NOT NULL THEN op.mls_desc
        ELSE 'NONE'
    END internal_name
FROM
    ols_property op WITH (NOLOCK)
    JOIN ols_listing ol WITH (NOLOCK) ON ol.fk_propid = op.pk_propid
WHERE
    op.city = :city
    AND op.fk_state = :state

SELECT
    ol.portal_eligible_bool
  , ol.trump_fk_propid
FROM
    ols_property op WITH (NOLOCK)
    JOIN ols_listing ol WITH (NOLOCK) ON ol.fk_propid = op.pk_propid
WHERE
    op.pk_propid = $propid

SELECT
    nm.pk_model_id
FROM
    nhc_model nm WITH (NOLOCK)
    LEFT JOIN ols_mls_supplier oms WITH (NOLOCK) ON oms.pk_supplier_id = nm.fk_supplier_id
    LEFT JOIN nhc_builder nb WITH (NOLOCK) ON nb.pk_builder_id = nm.fk_builder_id
    LEFT JOIN nhc_corporation nc WITH (NOLOCK) ON nb.fk_corporation_id = nc.pk_corporation_id
    LEFT JOIN ols_mls_supplier_extra omse_s WITH (NOLOCK) ON nm.fk_supplier_id =
omse_s.fk_supplier_id
        AND omse_supplier.search_code is null
    LEFT JOIN ols_mls_supplier_extra omse_c WITH (NOLOCK) ON nm.fk_supplier_id =
omse_c.fk_supplier_id
        AND omse_c.search_code = nc.bdx_id
```

```
WHERE
    NOT EXISTS (
        SELECT
            dbls.fk_model_id
        FROM
            ols.daily_builder_listing_stats dbls WITH (NOLOCK)
        WHERE
            nm.pk_model_id = dbls.fk_model_id
            AND TRUNC(dbls.imp_date) = TRUNC(SYSDATE - 1)
    )
    AND NVL(nm.is_hidden, 0) = 0
    AND nm.deleted IS NULL
    AND nm.is_model != 'Y'
    AND oms.active = 'Y'
    AND oms.hc_hide = 'N'
    AND (omse_corp.hc_hide_bool IS NULL OR omse_corp.hc_hide_bool = 0)
    AND (omse_supplier.hc_hide_bool IS NULL OR omse_supplier.hc_hide_bool = 0)

SELECT TOP (1)
    feedstarttime
FROM
    tblIDX_Feed_Listing_Source_History WITH (NOLOCK)
WHERE
    lsid = ?
```

Naming Conventions

Variables

Package variables should follow this format when referenced from outside the package (the full package name is inferred at declaration, so it is unneeded when declaring the variables in the package itself).

```
<POST-DASH REPO NAME>::<FILE NAME>::<variable name>
```

For example, If the repository is Common-Beanstalk and the file name is Interface.pm:

```
do_beanstalk_thing($BEANSTALK::INTERFACE::LastTubeUsed);
```

This is essentially a requirement when the package has a package namespace defined at the beginning of the file, and the variable name itself should be in CamelCase.

Variables in environmental files (typically environment.pm) maintain similar naming, but lowercase and without the filename:

```
$beanstalk::server_name = 'beanstalk11.dc3.homes.com';
```

Functions/Subroutines

Use descriptive, yet succinct, names.

Calls to subroutines should have no space between the name and opening parenthesis, and no spaces after opening or before closing parentheses.

```
my_sub('value1' => $value);
```

Current standard for passing parameters

Parameters are to be passed by name:

```
      f1(
          'p1' => 123,
          'P2' => 456,
          ...
      );
```

Functions will retrieve parameters via the following construct:

```
      sub f1 {
            %args = @_;
            ...
            if ($args{p1} == 123) { ... };
      }
```

Current standard for return values

Small functions with simple return values (usually boolean) can return the value itself.

Normal to large functions should return a hash with the following elements:

- `rv` - The data returned by the function. This can be a scalar/hash/array/etc. Only set upon success.
- `err_msg` - If an error occurred, the associated error message
- `err_type` - If an error occurred, the associated error code
- `err_cls` - If an error occurred, the class of error (e.g. ERROR, WARNING, etc.)

```
      $rv = {
            rv => ...,
            err_type => ...,
            Err_msg => ...,
            err_cls => ...
      };
```

Current standard for test organization

Tests should be organized according to the primary file they are being written for. Repositories with multiple .t files should have a wrapper .t script to run all of the tests in a single batch.

Ex:

Tests for Fetch.pm in Pipeline-Bulk should be gathered in Pipeline-Bulk/t/Fetch.t

Commenting

TBD
Ed Note: Write programmer intent comments
Obvious comments unneeded
Current code does not have enough comments

Unit Testing

Comparison Methods:

The use of specific test comparison methods from `Test::More` & `Test::Deep` will be based on the value's type (scalar, hash, array).

is(): Use of this method is restricted to only when comparing scalar values. Any use of this method when comparing hash & array values requires performing a separate comparison within or before the verification that prevents useful debugging information from being displaying upon a test failure.

Code Ex:

```
my $result = ‘Foo’;
my $expected = ‘Bar’;
is($result, $expected, ‘Testing Foo eq Bar’);
```

Failed Results Ex:

```
not ok 1 - Test_Foo_Bar
#   Failed test 'Test_Foo_Bar'
#   at example.t line 10.
#          got: ‘Foo’
#     expected: ‘Bar’
```

is_deeply(): Use of this method is for comparing hash & array values that meet the following criteria:

- The hash contains no arrays or contains arrays that will always result in the data falling into the same sort order.
- The array & any stored array will always result in the data falling into the same sort order.

Code Ex:

```
my $result = {
      scalar => ‘Foo’,
      array => [
            ‘Hello’,
            ‘World’
      ],
		hash => {
			new_value => 'Bar'
		}
	};

	my $expected = {
		scalar => 'Foo',
		array => [
			'Hello',
			'World'
		],
		hash => {
			new_value => 'Failed'
		}
	};

	is_deeply($result, $expected, 'Testing Hash');
```

Failed Results Ex:

```
	not ok 1 - Test Hash
	#   Failed test 'Test Hash'
	#   at example.t line 10.
	#     Structures begin differing at:
	#              $got->{hash}{new_value} = 'Bar'
	#     $expected->{hash}{new_value} = 'Failed'
```

cmp_deeply(): Use of this method is for comparing hash & array values that meet the following criteria:

- The hash contains array(s) values that can not be guaranteed to always fall into the same sorting order.
- The array & any array(s) contained within can not be guaranteed to always fall into the same sorting order.

This method requires the use of the `bag()` method on all arrays in the expected value to perform the comparison ignoring array order.
There are also various wrapper methods for the `cmp_deeply()` method that will also provide the same expanded output upon a test's failure. You may read up on those methods here:
http://search.cpan.org/~rjbs/Test-Deep-1.126/lib/Test/Deep.pm

Code Ex:

```
	my $results = {
		array => [1, 2, 3, 4],
		value => 'Foo',
		value_two => 'Bar'
	};
```

```
my @expected_array = (5, 2, 3, 1);
my $expected = {
      array => bag(@expected_array),
      value => ‘Foo’,
      Value_two => ‘Bar’
};

cmp_deeply($results, $expected, ‘Test Array’);
```

Failed Results Ex:

```
not ok 1 - Test Array
#   Failed test 'Test Array'
#   at example.t line 10.
# Comparing $data->{array} as a Bag
# Missing: 4
# Extra: 5
```

The Book of Python

Overall Standards

We will be using PEP8 as the underlying standard in all of our code. This document serves the purpose of filling in the holes where the PEP8 documentation is either bare or unclear. However, it is important to know that the specific examples listed in the following sections take precedence over any PEP8 standard.

The complete PEP8 documentation can be found here.

Miscellaneous Information

- Interface class files for workers should contain only static methods, with the instance for each worker in the Worker class file.

Importing Modules

Importing modules should be done in a specific manner.

Local environmental files:

```
    import Package.Module.environment as env
```

Non-local environmental files:

```
    import Package.OtherModule.environment as othername_env
```

Importing an Interface class from file:

```
    from Package.Module.Interface import Interface as module
```

Importing multiple non-Interface classes from a single file:

```
    from Pipeline.Common.Worker import BaseWorker, WorkerError
```

Importing an entire module (same as environmental files):

```
    import Common.Exceptions as exceptions
        called as >> exceptions.ExampleError
```

Modules should be lowercased, static and instance classes should be capitalized

```
    MODULES:
       import Common.Logging.Exceptions as logging_exceptions
       import Common.Exceptions.Interface as exceptions


    CLASSES:
        from Pipeline.Common.Worker import BaseWorker, WorkerError
        from Common.Logging.Interface import Interface as logging
```

Each module that has exceptions specific to that module should also be imported

```
    from Common.Logging.Interface import Interface as logging
    import Common.Logging.Exceptions as logging_exceptions
```

Python modules should be placed above our custom modules, separated by a newline, append path, newline

```
    import pytest
    import logging
    import sys

    sys.path.append('/code')

    from Pipeline.Common.Worker import BaseWorker, WorkerError
    from Common.MessageQueue.Interface import Interface as messagequeue
```

A requirements.txt file should be included for all NON-default python methods that are required to be installed via pip

```
% cat requirements.txt
>  pytest
>  mock
>  psutil
```

PEP8 Deviations

This section points out deviations from PEP8 standards for various reasons:

- **Line lengths** can be over the 80 character limit, 120 characters is a general guideline, but not a hard rule.
- **Comments** do not need to be complete sentences, this is often overly verbose for what is required.

Exceptions/Error Handling

Exceptions should be risen in the event that an error occurs in a given module. All custom exceptions should inherit from the *BaseError* class in Common-Exceptions

There are three types of exceptions we will be handling.

1. Common errors that any module at any time might throw or handle
    - Ex:MissingParameterError
2. Errors that are module specific and are handled inside a module by that same module
    - Ex: ClarifaRequestiFailed - in this case only the imagesearch worker will ever raise or catch this error
3. Errors that are from a module imported into another module
    - Ex: 'DBITimeoutError` - in this case any module that imports Common-Database could potentially catch this error, but would only raise it's own exception

- Say a worker calls MCP, but otherwise has no need for a database connection. The worker should only import MCP, and the MCP exceptions. The MCP module will then handle any errors thrown from Common-Database and return then as MCP exceptions to the worker which will then handle them accordingly.

Scenario 1 will require an addition to the Common-Exceptions package
Scenarios 2 and 3 will be added to the specific module they are related to

Function returns

Returns from functions should only contain useful information, i.e. there should be no trival True/False returns from a function if it is not needed. This is up to the best judgement of the developer as to whether or not a success case should be returned. For example:

```
    # Setting a value is non-volatile and no T/F return is necessary
    def set_value(self, new_value):
        self.value = new_value

    # Sending a RabbitMQ job is likely to fail, an exception should be thrown on
failure
    def send_to_rabbit(job):
        result = send(job)
        if result is FAILURE:
            raise Exception

    # A useful T/F value can be returned if needed
    def is_value_set(value):
        if value in not None:
            return True
        else:
            return False
```

If/Else Formatting

elif and else statements should start on their own line following their related if
Note: parenthesis should only be included in the case where it removes obfuscation

```
Parenthesis not needed
if a == 1:
    <code>
```

```
else:
   <code>

Parenthesis needed
if ((a or b) and f not in c) or (a is b):
    <code>
```

Also binary operators should follow the rule that words compare words, symbols compare numbers
For example:

```
Numbers:
     a == 2, b > 7
Words
    a is not None, b is True, c is not ‘string’
```

Dictionary Formatting

When assigning setting and accessing a dictionary we should be using the built in functions over bracket assessors whenever possible. For example:

```
value = dict_variable.get(‘key_one’) # This is the preferred format
dict_variable.update(key_two=2) # This is the preferred format
dict_variable.pop(‘key_three’) # This is the preferred format

value = dict_variable[‘key_one’] # This is should be avoided
dict_variable[‘key_two’] = 2 # This is should be avoided
del dict_variable[‘key_three’] # This is should be avoided
```

When updating dictionaries, the key-value combination should be squashed into single lines. In the case where these lines exceed the max line length they should be split into equal length lines

```
function_name({‘key_name’: other_dictionary.get('batch_sizes_time', 0) +
              (time.time() - batch_start)}) + more_stuff +
              and_some_other_final_things
```