PyDaF 0.3.0 manual

Emmanuel Benoît (ebenoit@ebenoit.info)

June 22, 2009

Contents

Chapter 1

Introduction

PyDaF stands for PYthon DAta Flows. It is a Python package that implements data-driven processing pipelines.

While programming usually involves writing a sequence of instructions, PyDaF instead revolves around two major elements: data units, which carry actual data as well as some associated metadata, and processors, which match data units depending on their type and associated metadata before processing them.

It is up to the programmer to define data types and processor input specifications; however, the library will then automatically handle finding matching data units, locking them and running the appropriate processors.

The library is also capable of running multiple processors in parallel. For now, this can only be done locally, using threads. However, running PyDaf processing pipelines on multiple computers will be possible in the future.

Features

- Written in pure Python. Version 2.6 is required.
- Synchronous or threaded execution modes.
- Support for multiple jobs.
- Automatic parallelisation of tasks.
- Data unit wrappers for common Python types.
- MapReduce-like algorithms can be easily implemented.
- Dead ends in a processing pipeline are detected automatically.
- Exceptions in processors are caught and stored as the cause for a job's failure.

A note about version numbers

PyDaF is currently in a highly unstable state, as it is still under development. However, the version numbers will follow the scheme described below.

- The first number corresponds to the stable release number (0 at the moment, as no stable versions have been released).
- The second number corresponds to the current API; a stable version will always use 0 here.
- The third number corresponds to a patch level; no incompatibilities with other patch levels should be present.

Copyright and license

1.1 Concepts

In order to understand and use the PyDaF library, it is necessary to know the different elements it manipulates. This section describes these various elements.

Data units Data units are the most basic elements manipulated by PyDaF. A data unit is an instance of a class that includes both actual data and PyDaF-specific metadata. The most important part of a data unit's metadata is the set of *flags*. A flag is a short string that can be associated to a data unit, indicating that it has undergone some transformation. Other metadata include the data unit's lifetime (an integer which is decreased when a data unit is processed and must never reach 0 - it prevents a job from entering an endless loop), as well as its modification or retirement status.

Data descriptions A data description consists in a type, as well as two optional elements: a set of flags the data being described must have, and a set of flags the data being described must not have. They can be used to retrieve sub-collections of specific data units from collections, or as part of data specifications.

Collections A collection is a set of data units. The metadata of the data units in a collection can be manipulated *en masse*. It is also possible to create named sub-collections that automatically match some descriptions, or to retrieve parts of the data units using the descriptions directly.

Data specifications Data specifications are very similar to data descriptions, but they also include indications about the cardinality of a set of data units. They are used to specify a processor's input.

Processors Processors are the active elements of a PyDaF processing pipeline. They are executed when data units matching their input specifications are found.

Specificity Descriptions, specifications and processors can be more or less specific. For example, a description matching the *Integer* data type is less specific than a description matching the *Integer +someflag* data type. A specification's specificity is computed from the description it encapsulates and from the cardinality it indicates, while a processor's specificity is computed from its various input specifications.

Pipelines, jobs and tasks A pipeline is a set of processors associated with a method of execution. A job is a top-level execution element. It is executed through a pipeline from some specific input data units and runs until either a dead-end is found, an exception is raised or a specific set of data units is produced. A task is a part of a job's execution - it associates a processor with specific data units.

1.2 Installation

The PyDaF distribution archive includes a `setup.py` script based on Python's distutils. Running the `python setup.py install` command will install the PyDaF package automatically. However, the documentation and examples will not be installed.

The following sub-directories can be found in the PyDaF distribution:

- `pydaf/` contains the PyDaF package itself. If you do not use the supplied installation script, you will have to install it manually. Install this directory somewhere in your Python search path. On most systems, you can use the `lib/site-packages` directory. Its exact location might vary depending on your specific Python installation. Alternatively, you may copy it to an arbitrary location and manually add this location to your Python search path (e.g. in the `PYTHONPATH` environment variable).
- `doc/` contains the TeX source for this documentation. You may use the file as-is, or compile it to a PDF, or generate HTML documentation from it.
- `examples/` contains a few examples.

Chapter 2

Programming with PyDaF

There are two main aspects to programming using the PyDaF library - describing data types, and writing processors. However, running a processing pipeline is a separate process which needs to be described in its own right. We will also have a look at the MapReduce helper class, which allows easy implementation of MapReduce-like sets of processors, before giving a full example.

2.1 Defining data types

Before we start, it is important to mention that PyDaF data types always correspond to Python classes. However, relationships between Python classes are completely ignored by PyDaF - a class that inherits another is considered as a completely different type.

There are two possible ways of defining PyDaF data types. The simplest, which is recommended in most cases, is to use PyDaF's `pydaf.TypeWrapper` class to create a wrapper for an already existing type. However, if that is required, it is also possible to define custom classes and use them as PyDaF data types directly through the more generic `pydaf.Data` class.

2.1.1 PyDaF types from existing Python types

Wrapper definition

When used as a class's metaclass, `pydaf.TypeWrapper` will look for information about the existing Python type from the new class's `Meta` member, which must also be a class. This inner class should contain all information pertaining to the data type to be used, and may include some PyDaF-specific information such as the data type's default lifetime and default flags.

Of course, the principal information required when a wrapper is being created is the type itself. This information is given through the `std_type` attribute.

Operators and methods can also be wrapped. Here, by *operator*, we mean to indicate some internal method with a name matching the `__name__` pattern, returning some value of the type being wrapped, and which, when being called, will not modified the value. *Methods*, on the other hand, can be just any method.

To instruct PyDaF to create wrappers around methods, the `operators` attribute must be present in the `Meta` class; it must be an iterable that contains

the names of the various operators for which a wrapper is to be created.

Method wrappers are listed as the keys of a dictionary, to which a boolean value is associated. This value indicates whether calling the method in question will modify the value or not. This information is important as it will allow a data unit's metadata to be properly updated when the method is called.

Warnings and notes

- Do not wrap comparison operators or the `__hash__` method through the wrapper class, as the resulting data units would no longer be compared on the basis of being data units, but as values of the type they are supposed to wrap.
- Do not attempt to write an `__init__` method, as it will be replaced by the metaclass.
- Other attributes may be used in the `Meta` class. These attributes are the same than the ones used for custom types.

```
import pydaf

class Unicode( object ):
    __metaclass__ = pydaf.TypeWrapper
    class Meta:
        std_type = unicode
        operators = ( 'add' , 'mul' , 'getslice' )
        methods = {
            '__repr__'  : False ,
            '__str__'   : False ,
            'count'     : False ,
        }
```

Figure 2.1: Definition of a partial wrapper around the standard Unicode string type.

PyDaF wrappers

PyDaF includes wrappers for some of Python's types. The following wrappers can be found in the `pydaf.std_types` module:

- `Integer`: wrapper around the `int` type,
- `Float`: wrapper around the `float` type,
- `String`: wrapper around the `str` type,
- `List`: wrapper around the `list` type,
- `Set`: wrapper around the `set` type,
- `Dict`: wrapper around the `dict` type.

Using wrapped data units

Initialisation Wrapped data units can be initialised in two different ways:

- either by calling the constructor with no arguments - this will only work if the wrapped type supports an argument-less constructor,
- or by calling the constructor with a value of the specified type as its parameter.

In the latter case, please be aware of the fact that the value will not be copied. If it is a reference to an object, modifying the object will affect the data unit.

Operators and methods Operators and methods can be called transparently; if they are used with wrapped data units as their parameters, their values will be automatically extracted and passed to the original method or operator.

```
v1 = pydaf.std_types.String( "abc" )
v2 = pydaf.std_types.String( "def" )
v3 = v1 + v2
print v3
v4 = v1[ 1: ]
print v4
print v3.count( "a" )
```

Figure 2.2: Using wrapped operators and methods

Accessing the value directly The value of a wrapped data unit can always be accessed directly through the instance's `value` attribute. However, modifying the value directly will not automatically update the modification status of the data unit; the unit's `pydaf.modified` metadata must be set to `True` manually. This issue will be addressed in a future version of the library.

```
# "v" contains an Integer data unit
v.value = v.value + 1
v.pydaf.modified = True
```

Figure 2.3: Modifying a wrapped data unit's value

2.1.2 Creating types manually

While using wrappers for existing Python types is sufficient under most circumstances, it is sometimes more convenient to create custom types which implement their own, specific behaviours. In these cases, the `pydaf.Data` class can be used as the new class's metaclass; this will result in the creation of a wrapper around the `__init__` and `__repr__` methods (or, if they are not present in the

new class, they will be created) in order to implement the initialisation of the associated metadata.

Manually-created data types may contain a `Meta` class which defines some defaults for the data units' metadata:

- the `default_lifetime` attribute defines the data unit's maximal lifetime,
- the `default_flags` attribute must contain an iterable that will list the flags a data unit should possess just after it has been created.

```
import pydaf

class Counter( object ):
    __metaclass__ = pydaf.Data

    class Meta:
        default_flags = ( 'counting' , )

    def __init__( self , value = 100 ):
        assert( value >= 1 )
        self.counter = value

    def decrease( self ):
        assert( self.counter > 0 )
        self.counter = self.counter - 1
        self.pydaf.modified = True
        if self.counter == 0:
            self.pydaf.replace_flag( 'counting' , 'done' )
```

Figure 2.4: Creating a custom "counter" type

2.2 Manipulating data units

While data units themselves act according to their type's definition, manipulating them also includes accessing their metadata. In addition, PyDaF includes a `Collection` class that allows sets of data units to be manipulated in various ways.

2.2.1 Accessing data units' metadata

A data unit's metadata can be accessed through the `pydaf` attribute. This attribute is actually an instance of the `Metadata` class, which holds the unit's lifetime, its flags, and its retirement and modification status.

Checking for a flag The `has_flag()` method allows its caller to check whether a data unit has a specific flag or not. Its parameter is a string that contains the flag's name, and it will return a boolean.

```
if c.pydaf.has_flag( 'done' ):
    # Do something.
    pass
```

Figure 2.5: Checking for the presence of a flag

Setting and clearing flags Setting and clearing flags can be accomplished through the `set_flag()` and `clear_flag()` methods, respectively. Both methods take the flag's name as their parameters. When trying to set a flag that is already present or, conversedly, trying to clear a flag that is not present, the `ValueError` exception will be raised. Both methods can be chained, as they return a reference to the `Metadata` instance they belong to.

```
c.pydaf.set_flag( 'flag0'
      ).set_flag( 'flag1'
      ).clear_flag( 'flag2' )
```

Figure 2.6: Setting and clearing flags

One common operation is to replace a flag with another. Instead of using both methods, it is possible to call the `replace_flag()` method instead, with the first argument being the flag to be replaced and the second being the new flag. If the original flag is not present or if the new one already is, the `ValueError` exception will be raised.

Modification status When a data unit is being modified inside a processor, it is important for PyDaF to be informed of the change. The `modified` field of the metadata object should be set to `True` in these cases.

Retirement Data units can be "retired" - that is, when all processors are done handling a specific data unit, this data unit will be removed from the job's data pool and no further attempts to match it with the processors' data input specifications will be made. This can be accomplished through the metadata instance's `retire()` method.

2.2.2 Data unit descriptions

Before we go any further, we must look at the way data units can be described. The `pydaf.Description` class can be used to store data unit descriptions. Its constructor has a mandatory parameter, which should be the type of the data units. In addition, two named parameters may be used, `flags` and `not_flags`; these parameters should either be `None` (which is the default), or some iterable containing the names of the flags matching data units should or should not have, respectively.

Data unit descriptions have a `match` method which, if called with a data unit as its parameter, will check if the data unit matches the description, returning

a boolean accordingly.

```
from pydaf.std_types import Integer, String
from pydaf import Description as D

d = Integer( )
d.pydaf.set_flag( 'test' )

D( Integer ).match( d )
# -> True
D( Integer , flags = ( 'test' , ) ).match( d )
# -> True
D( String ).match( d )
# -> False
D( Integer , not_flags = ( 'test' , ) ).match( d )
# -> False
```

Figure 2.7: Data unit descriptions

Specificity The specificity of a description is computed using the following formula: $specificity = 1 + 4 * len(flags) + 2 * len(not\_flags)$

2.2.3 Collections

Collections are selectable sets of data units. The data units in a collection may be filtered using descriptions. In addition, the metadata manipulation methods described above can also be used on entire collections, although their behaviour is slightly different. It is also possible to create named filters on a collection, allowing the filtered data to be accessed easily.

Creating and updating collections Collections are implemented through the `pydaf.Collection` class. Its constructor takes an optional argument, which should be either a data unit, an iterable containing data units or another collection. If the argument is not present, the collection will be initially empty; otherwise the argument's value (if it is a data unit) or its contents (if it is an iterable or a collection) will be added to the collection.

Once created, collections can be updated through the `add` and `remove` methods. Both methods must be called with an argument that is either a data unit, an iterable containing data units or a collection. They will respectively add or remove the specified data unit(s) from the collection.

Filtering collections Data units listed in a collection can be filtered using the `filter` and `exclude` methods. Both methods take any amount of data unit descriptions as their arguments, and will return a new collection containing data units matching one of the descriptions or matching none of the descriptions, respectively.

```
import pydaf
from pydaf.std_types import Integer, String
from pydaf import Description as D

c = pydaf.Collection([
    Integer( 0 ) , String( 'a' ) , String( 'b' )
])

c.filter( D( Integer ) )
# [ Integer( 0 ) ]
c.exclude( D( Integer ) )
# [ String( 'a' ) , String( 'b' ) ]
```

Figure 2.8: Filtering collections

Named sub-collections It is possible to create automatic, named filters otherwise known as sub-collections. These filters will be applied automatically when the corresponding attribute is accessed.

Named sub-collections are created by calling a collection's `sub_collection` method. The first argument should be the name of the new sub-collection, and this argument should be followed by at least one description. If a sub-collection with the same name already exists, the `KeyError` exception will be raised. Once created, a sub-collection can be accessed through the attribute corresponding to its name in the parent collection.

Important — Sub-collections should not be modified in any way. The changes will not propagate to the parent collection.

```
import pydaf
from pydaf.std_types import Integer, String
from pydaf import Description as D

c = pydaf.Collection([
    Integer( 0 ) , String( 'a' ) , String( 'b' )
])

c.sub_collection( 'strings' , D( String ) )
c.strings
# [ String( 'a' ) , String( 'b' ) ]
```

Figure 2.9: Named sub-collections

When a new collection is created from another, the original collection's named sub-collections are copied over. When a collection that contains named sub-collections is added to another using the `add` method, its sub-collections are re-created in the target collection, unless a sub-collection using the same name is already present (in which case the sub-collection is simply ignored, no

exception is raised).

Other access methods Collections provide a few other access methods. First, collections are iterable, and can be used in `for` loops or in generators. The `in` operator is also supported to allow checking for the presence of a specific data unit. Support for `len` is also present. Finally, collections support two different types of subscripting:

- using an integer, which will return the value found at the specified index,
- using a data description, which will return a tuple of data units matching the description.

Metadata Collections allow their contents' metadata to be manipulated directly. The following methods are available:

- `has_flag()` will return `True` if all the units in a collection have the specified flag,
- `set_flag()`, `clear_flag()` and `replace_flag()` are supported; they will not cause exception to be raised, and will instead return the amount of data units which were affected by the specified modification.
- `retire()` may be called to mark all of a collection's data units as retired.

2.2.4 Specifications

Creating specifications Specifications are written using the `pydaf.Specification` class. Like data unit descriptions, the constructor of this class requires a first argument that indicates the type of the data units to match. The `flags` and `not_flags` arguments are also supported.

In addition, a specification may indicate the matched units' cardinality using the `min_count` and `max_count` arguments. If these arguments are not present, they will both default to 1. `min_count` must be at least 1 (it is impossible to use completely optional data unit matches). `max_count` can be any integer greater or equal to `min_count`'s value, or it can be `None` if all matching data units are to be used.

Finally, specifications may also contain a `to` argument, which will be used to create named sub-collections automatically in the resulting collections.

Specifications for multiple types of data can be combined using the `&` operator.

Matching specifications Specification instances have a `match` method which takes a collection as its argument. If the collection doesn't contain data units matching the whole specification, the `Specification.NoMatch` exception will be raised. Otherwise, a new collection will be returned; this new collection will contain the matched data units and, if any part of the specification was using the `to` argument, it will have the corresponding sub-collections. When the collection contains more data units matching a part of the specification than the maximal cardinality indicates, matching data units will be selected at random.

Specificity A single specification's specificity is computed from the corresponding description's specificity. If there is no maximal cardinality, the specification and the description have the same specificity; however, if there is a maximal cardinality, this value is multiplied by 2. In the case of combined specifications, the average of its elements' specificity is used.

```
import pydaf
from pydaf.std_types import Integer, String
from pydaf import Specification as S

s1 = S( Integer ) & S( String )
s2 = S( Integer , flags = ( 'test' , ) ) & S( String )
c = pydaf.Collection([
    Integer( 0 ) , String( 'a' ) , String( 'b' )
])

s1.match( c )
# returns either
# [ Integer( 0 ) , String( 'a' ) ]
# or
# [ Integer( 0 ) , String( 'b' ) ]
s2.match( c )
# Will raise Specification.NoMatch
```

Figure 2.10: Creating and matching specifications

2.3 Defining processors

While PyDaF data units represent the data that is being manipulated, processors are responsible for handling the data. Processors are written as classes; they must inherit the `pydaf.Processor` class.

Writing a PyDaF processor can be seen as a two-step process:

- first, the data units the processors will match must be defined,
- then, the processor's actual code must be written.

2.3.1 Processor input

Processors must include a `Meta` class which will contain information about the processor's input. Two attributes can be specified, and while only one of them will suffice, it is mandatory for a processor to contain at least one of them.

- The `input` attribute should contain the specification for a processor's read-only input. Data units matching the specification will be used in the processor, but may also be used by other processors at the same time.

- The `inout` attribute should contain the specification for data units the processor must match but that it is also capable of modifying. These data units will be locked exclusively for a processor instance.

Specificity A processor's specificity can be computed from these two attributes; the read/write input specification is considered much more specific than the read-only input specification. The following formula is used:

$specificity(processor) = 1 + 20 * specificity(inout) + specificity(input)$

A processor's specificity is very important, as it will determine the order in which the processors' input data units are looked up. The more specific a processor, and the earlier its input will be looked up.

2.3.2 Processing method

Actual processing is implemented in a processor's `process` method, which doesn't support any arguments with the exception of the processor's instance. However, when the method is called, data units selected as input will be available through `self.input` and `self.inout` (for RO and RW input collections, respectively). In addition, `self.output` will be an empty collection to which data units generated by the processor may be added.

Guidelines

- Processors should be kept short and simple.
- Processors should never use any data that isn't part of their input data units.
- Exceptions raised inside a processor, either on purpose or due to programming errors, will cancel the whole job.
- While it is actually possible to modify data units from the `self.input` collection, it should never be done as these data units will not be properly handled by the pipeline; in addition, read-only input may be shared between different processors.
- The only exception to this rule is using the `retire()` method, as it will not affect other processors and will still be applied once the data unit is not longer locked by any processor.
- When a data unit has been processed by a given processor type, it will not be re-processed again unless it is modified and still matches the processor's input specifications. This is potentially dangerous as a modified data unit will be reprocessed until its lifetime runs out if the specifications allow it.
- Each time a data unit is processed, its lifetime is decreased.
- The lifetime of data units produced by a processor is set to the lowest lifetime in the input collections.

```
import pydaf
from pydaf.std_types import Integer
from pydaf import Specification as S
from pydaf import Description as D

class AddNumbers( pydaf.Processor ):

    class Meta:
        input = S( Integer ,
                   flags = ( 'add-me' , ) ,
                   min_count = 2 ,
                   max_count = None )

    def process( self ):
        sum = reduce( lambda x,y: x + y ,
                      self.input[ D( Integer ) ] )
        sum.pydaf.set_flag( 'result' )
        self.output.add( sum )
```

Figure 2.11: A simple processor that adds numbers

2.4 Running PyDaF code

PyDaF code is executed using pipelines. A pipeline is basically a collection of processors and encapsulates the code required to manage their execution in the proper order.

The PyDaF library provides two kinds of pipelines:

- synchronous pipelines run in the calling thread; they never parallelise task execution and, when a job is requested, it is processed immediately,
- threaded pipelines run in a separate thread; they use *runners* to execute the various tasks. It is possible to create both completely multithreaded, parallelising pipelines or pipelines that will only execute one task at a time directly inside the pipeline's thread.

2.4.1 Using the synchronous pipeline

Synchronous pipelines are the simplest kind of pipeline; they are implemented by the `pydaf.SynchronousPipeline` class.

Once created, a synchronous pipeline must be assigned a set of processors. This is accomplished using the `add_processors()` method which takes any number of processors or iterables containing processors as its arguments. A pipeline that contains at least one processor is initialised and ready to be used.

Calling the pipeline's `process` method will cause a job to be processed. Two arguments are required:

- `input_data`, an iterable containing the job's input data;

- `output_specification`, a specification which, when matched, will end the processing.

The `process()` method returns an identifier which must then be used when calling the `get_result()` method. This method will either raise the `JobFailed` exception if the job failed; in this case, the exception's value will have a `cause` attribute, containing details about the exception which caused the job to fail, as provided by `sys.exc_info()`. Otherwise, a collection matching the output specification passed to `process()` is returned.

By default, `get_result()` will remove the job's result from the pipeline. However, it is possible to leave the data in the pipeline's care by passing an argument named `clear_result` to the method, using `False` as the value.

```
import pydaf
from pydaf.std_types import Integer

class AddNumbers( pydaf.Processor ):
    # ....

# Set up the pipeline
pipeline = pydaf.SynchronousPipeline( )
pipeline.add_processors( AddNumbers )

# We want to add 3 numbers...
input = [ Integer( 1 ) , Integer( 2 ) , Integer( 3 ) ]
wanted = S( Integer , flags = [ 'sum' ] )

# Start the process then get the result
result_id = pipeline.process( input , wanted )
try:
    result = pipeline.get_result( result_id )
except pydaf.JobFailed , exc:
    from traceback import print_exception
    print
    print "JOB FAILED"
    print_exception( *( exc.cause ) )
    print
    sys.exit( -1 )
```

Figure 2.12: Using a synchronous pipeline

2.4.2 Using the threaded pipeline

The threaded pipeline is implemented by the `pydaf.ThreadedPipeline` class.

To initialise a threaded pipeline, it must be created and processors must be added using the `add_processors()` method (which is identical to the one used by synchronous pipelines). It must then be assigned a set of *runners*, which are used by the pipeline to execute tasks. Runners are added using the

`add_runner()` method, which takes a runner class as its first parameter. The second, optional argument is an integer that indicates the amount of instances that should be created from the specified runner type. Two types of runners are available at this point:

- `pydaf.BasicRunner` is a synchronous runner; it should only be used alone, as the pipeline's only runner. Using it will cause the pipeline to run the tasks in a sequence, making it nearly identical to a synchronous pipeline. Jobs will still be added and processed asynchronously, however.
- `pydaf.ThreadedRunner` is a runner that executes tasks in separate threads.

Once the pipeline is ready, its `process()` method can be called to start jobs. The method always returns immediately.

To obtain a job's result, it must first have been completed. If `get_result()` is called before that is the case, the `JobRunning` exception will be raised. Otherwise, the method behaves pretty much like the synchronous pipeline's. The `wait()` method, if called with a job identifier as its argument, will wait for the specified job to complete.

Finally, the `stop()` method can be used to terminate the pipeline's processing.

```
pipeline = pydaf.ThreadedPipeline( )
pipeline.add_runner( pydaf.ThreadedRunner , count = 8 )
pipeline.add_processors( Proc1 , Proc2 , Proc3 )

try:
    result_id = pipeline.process( ... )
    pipeline.wait( result_id )
    try:
        result = pipeline.get_result( result_id )
    except pydaf.JobFailed , exc:
        from traceback import print_exception
        print
        print "JOB FAILED"
        print_exception( *( exc.cause ) )
        print
        sys.exit( -1 )
finally:
    pipeline.stop( )
```

Figure 2.13: Using a threaded pipeline

2.5 MapReduce helper class

The `pydaf.MapReduce` class makes it easy to generate sets of processors that implement the Map/Reduce programming pattern. It must be inherited, with the child class containing the implementation of the splitting, mapping and

reducing parts of the code as well as the information regarding the data types and the general behaviour of the generated set of processors.

Note — no example is given in this section. However, the full example of chapter 3 contains a Map/Reduce implementation.

2.5.1 Metadata

A class that inherits `pydaf.MapReduce` must contain a class named `Meta` which will contain the various information defining the system's behaviour. The following attributes are supported.

- `name` is an optional string containing the base name of the system's processors and of the flags used to mark intermediary data units. If this attribute is not present, the class's name will be used.
- `in_type` is a specification that will be used to locate the system's input data.
- `output_flag` is a flag that will be set on the final `Dict` data unit generated by the reducing part of the system.
- `part_type` is the type of the data units that will be produced by the splitting code and expected by the mapping code.
- `retire_input` is an optional attribute indicating whether the data accepted by the splitting code should be retired. The data will not be retired if this attribute is not present.

2.5.2 Splitting

Splitting the input is implemented by overriding the `split()` method. When the method is called, it is given two arguments:

- `input` is the processor's actual input; it is read-only.
- `output` is a Python list to which the generated data units should be appended.

The processor that handles the splitting code will add the `split-NAME` flag to the generated data units; in addition, it will create an `Integer` data unit with the `counter-NAME` flag, containing the amount of data units generated by the splitting code, and a `Dict` data unit with the `incomplete-NAME` flag, which will later be used to store the final result.

2.5.3 Mapping

Mapping the various parts is implemented by overriding the `map()` method. This method needs to accept two parameters:

- `input`, which will contain the read-only data unit for the part being mapped,
- `output`, which will contain a `Dict` data unit.

The processor will call this method, then it will retire the input data unit and add the output data unit to the job's pool, with the `map-NAME` flag set.

2.5.4 Reducing

Reduction of the dictionaries produced by the mapping code into a single dictionary is implemented by overriding the `reduce()` method. This method accepts two arguments:

- `input` which contains a single dictionary produced by the mapping code,
- `output` which refers to the main output dictionary.

The `reduce()` method is called once for every part that has been processed. After it is called, the processor will retire the intermediary data units and decrease the counter's value. If the counter reaches 0, its data unit is retired and the main output dictionary gets its flags modified: the `incomplete-NAME` flag is removed, while the flag specified by the metadata is set.

Chapter 3

Full example

This chapter contains a full example of PyDaF's usage. The example implements a word counter using a Map/Reduce algorithm. However, it harnesses processor specificity to modify the parts generated by the splitting code before they are mapped. Finally, another processor removes all words that are less than 2 characters long from the generated dictionary.

We will use a threaded pipeline as it is more appropriate for Map/Reduce algorithms. The input of a job will be a `List` data unit with the `input-files` flag, containing the list of files from which words are to be counted. We will expect a `Dict` with the `result` flag to be produced as a result of the pipeline.

The complete source code for this example is available in the PyDaF distribution; it can be found in the `examples/specificity/` directory.

3.1 The Map/Reduce implementation

The first thing we will need is the Map/Reduce class; this class will implement the actual splitting and word counting of the input. It will retire its input data, as the list of files is no longer required after the files have been read. Parts will be strings containing 100 lines from an input file. In the end, the generated dictionary will have the `wordcount` flag.

First, we need to define the Map/Reduce implementation's metadata:

```
class WordCounter( pydaf.MapReduce ):

    class Meta:
        name = 'wcount'
        retire_input = True
        in_type = S( List ,
                     flags = ( 'input-files' , ) ,
                     to = 'files' )
        part_type = String
        output_flag = 'wordcount'
```

This being done, we will define the splitting code. We must be careful not to generate parts that contain no text at all - not that it would be a problem, but it would be useless.

```
def split( self , input , output ):
    max_len = 100
    for f_name in input.files[ 0 ]:
        f = open( f_name )
        try:
            f_split = [ ]
            got_it = False
            for line in f:
                got_it = True
                f_split.append( line )
                if len( f_split ) > max_len:
                    txt = String( ''.join( f_split ) )
                    output.append( txt )
                    f_split = []
            if got_it and len( f_split ):
                txt = String( ''.join( f_split ) )
                output.append( txt )
        finally:
            f.close( )
```

The resulting processor will generate a set of `String` units with the `split-wcount` flag set, as well as the incomplete output dictionary and parts counter.

The mapping code is simple enough:

```
def map( self , input , output ):
    for word in input.split():
        if word not in output:
            output[ word ] = 1
        else:
            output[ word ] = output[ word ] + 1
```

So is the reducing code, which basically adds all counts to the final dictionary.

```
def reduce( self , input , output ):
    for word , count in input.items( ):
        if word in output:
            count = count + output[ word ]
        output[ word ] = count
```

3.2 Pre-processing the parts

We will now define a processor that will take the parts generated by the splitting code above before they are mapped. This can be done easily by accepting the parts as a read/write input unit, and setting a new flag to make sure the parts are not processed twice.

The processor itself will make sure that all text is lower-case, and that it only includes actual words.

```
class PartPreprocessor( pydaf.Processor ):
```

```
    class Meta:
        inout = S( String ,
                   flags = ( 'split-wcount' , ) ,
                   not_flags = ( 'preprocessed' , ) ,
                   to = 'text' )

    def process( self ):
        t = self.inout.text[ 0 ]
        t.value = re.sub( r'[^a-z]' , ' ' , t.lower() )
        t.pydaf.set_flag( 'preprocessed' )
```

3.3 Final dictionary clean-up

A last processor is required to remove all words shorter than 2 characters from the dictionary.

```
class ShortWordRemover( pydaf.Processor ):

    class Meta:
        inout = S( Dict ,
                   flags = ( 'wordcount' , ) ,
                   to = 'wc' )

    def process( self ):
        d = self.inout.wc[ 0 ]
        for word in d.keys( ):
            if len( word ) < 3:
                del d[ word ]
        d.pydaf.replace_flag( 'wordcount' , 'result' )
```

3.4 Putting it all together

Now that the processors have been defined, the only thing left to do is to create a pipeline and have it process the jobs. First, we will initialise the pipeline:

```
pipeline = pydaf.ThreadedPipeline( )
pipeline.add_runner( pydaf.ThreadedRunner , count = 8 )
pipeline.add_processors( WordCounter( ) ,
                         PartPreprocessor ,
                         ShortWordRemover )
```

In this example, we use 8 concurrent threads. The pipeline is initialised using the processors generated by the Map/Reduce helper as well as the two manually defined processors.

Once this is done, we will need to generate the input data. We will use the script's arguments as the initial contents of the list.

```
l = List( sys.argv[ 1: ] )
l.pydaf.set_flag( 'input-files' )
```

Once this is accomplished, we still need to run the job:

```
wanted = S( Dict , flags = [ 'result' ] )
try:
    result_id = pipeline.process( [ 1 ] , wanted )
    pipeline.wait( result_id )
    try:
        result = pipeline.get_result( result_id )
    except pydaf.JobFailed , exc:
        from traceback import print_exception
        print
        print "JOB %d FAILED" % result_id
        print_exception( *( exc.cause ) )
        print
        sys.exit( -1 )

    for word , count in result[0].items( ):
        print "'%s': %d" % ( word , count )
finally:
    pipeline.stop( )
```