

---

# ruffus Documentation

*Release 2.4.2*

**Leo Goodstadt**

July 08, 2014



CONTENTS



**START HERE:**

## 1.1 Installation

Ruffus is a lightweight python module for building computational pipelines.

### 1.1.1 The easy way

*Ruffus* is available as an `easy-install`-able package on the Python Package Index.

```
sudo pip install ruffus --upgrade
```

This may also work for older installations

1. Install setuptools:

```
wget peak.telecommunity.com/dist/ez_setup.py
sudo python ez_setup.py
```

2. Install *Ruffus* automatically:

```
easy_install -U ruffus
```

### 1.1.2 The most up-to-date code:

- Download the latest sources or
- Check out the latest code from Google using git:

```
git clone https://bunbun68@code.google.com/p/ruffus/ .
```

- Bleeding edge Ruffus development takes place on github:

```
git clone git@github.com:bunbun/ruffus.git .
```

- To install after downloading, change to the , type:

```
python ./setup.py install
```

## Graphical flowcharts

**Ruffus** relies on the `dot` programme from **Graphviz** (“Graph visualisation”) to make pretty flowchart representations of your pipelines in multiple graphical formats (e.g. `.png`, `.jpg`). The crossplatform Graphviz

package can be [downloaded here](#) for Windows, Linux, Macs and Solaris. Some Linux distributions may include prebuilt packages.

**For Fedora, try**

```
yum list 'graphviz'
```

**For ubuntu / Debian, try**

```
sudo apt-get install graphviz
```

## 1.2 Ruffus Manual: List of Chapters and Example code

Download as pdf.

- **Chapter 1:** *An introduction to basic Ruffus syntax*
- **Chapter 2:** *Transforming data in a pipeline with @transform*
- **Chapter 3:** *More on @transform-ing data*
- **Chapter 4:** *Creating files with @originate*
- **Chapter 5:** *Understanding how your pipeline works with pipeline\_printout()*
- **Chapter 6:** *Running Ruffus from the command line with ruffus cmdline*
- **Chapter 7:** *Displaying the pipeline visually with pipeline\_printout\_graph()*
- **Chapter 8:** *Specifying output file names with formatter() and regex()*
- **Chapter 9:** *Preparing directories for output with @mkdir*
- **Chapter 10:** *Checkpointing: Interrupted Pipelines and Exceptions*
- **Chapter 11:** *Pipeline topologies and a compendium of Ruffus decorators*
- **Chapter 12:** *Splitting up large tasks / files with @split*
- **Chapter 13:** *@merge multiple input into a single result*
- **Chapter 15:** *Logging progress through a pipeline*
- **Chapter 14:** *Multiprocessing, drmaa and Computation Clusters*
- **Chapter 16:** *@subdivide tasks to run efficiently and regroup @collate*
- **Chapter 17:** *@combinations, @permutations and all versus all @product*
- **Chapter 18:** *Turning parts of the pipeline on and off at runtime with @active\_if*
- **Chapter 20:** *Manipulating task inputs via string substitution with inputs() and add\_inputs()*
- **Chapter 19:** *Signal the completion of each stage of our pipeline with @posttask*
- **Chapter 21:** *Esoteric: Generating parameters on the fly with @files*
- **Chapter 22:** *Esoteric: Running jobs in parallel without files using @parallel*
- **Chapter 23:** *Esoteric: Writing custom functions to decide which jobs are up to date with @check\_if\_upToDate*
- **Appendix 1** *Flow Chart Colours with pipeline\_printout\_graph*
- **Appendix 2** *Under the hood: How dependency works*

- [Appendix 3 Exceptions thrown inside pipelines](#)
- [Appendix 4 Names \(keywords\) exported from Ruffus](#)
- [Appendix 5: Legacy and deprecated syntax @files](#)
- [Appendix 6: Legacy and deprecated syntax @files\\_re](#)

**Ruffus Manual:** List of Example Code for Each Chapter:

- [Chapter 1: Python Code for An introduction to basic Ruffus syntax](#)
- [Chapter 1: Python Code for Transforming data in a pipeline with @transform](#)
- [Chapter 3: Python Code for More on @transform-ing data](#)
- [Chapter 4: Python Code for Creating files with @originate](#)
- [Chapter 5: Python Code for Understanding how your pipeline works with pipeline\\_printout\(...\)](#)
- [Chapter 7: Python Code for Displaying the pipeline visually with pipeline\\_printout\\_graph\(...\)](#)
- [Chapter 8: Python Code for Specifying output file names with formatter\(\) and regex\(\)](#)
- [Chapter 9: Python Code for Preparing directories for output with @mkdir\(\)](#)
- [Chapter 10: Python Code for Checkpointing: Interrupted Pipelines and Exceptions](#)
- [Chapter 12: Python Code for Splitting up large tasks / files with @split](#)
- [Chapter 13: Python Code for @merge multiple input into a single result](#)
- [Chapter 14: Python Code for Multiprocessing, drmaa and Computation Clusters](#)
- [Chapter 15: Python Code for Logging progress through a pipeline](#)
- [Chapter 16: Python Code for @subdivide tasks to run efficiently and regroup with @collate](#)
- [Chapter 17: Python Code for @combinations, @permutations and all versus all @product](#)
- [Chapter 20: Python Code for Manipulating task inputs via string substitution using inputs\(\) and add\\_inputs\(\)](#)
- [Chapter 21: Esoteric: Python Code for Generating parameters on the fly with @files](#)

## 1.3 Chapter 1: An introduction to basic *Ruffus* syntax

See also:

- [Manual Table of Contents](#)

### 1.3.1 Overview



Computational pipelines transform your data in stages until the final result is produced. One easy way to understand pipelines is by imagining your data flowing across a series of pipes until it reaches its final destination. Even quite complicated processes can be broken into simple stages. Of course, it helps to visualise the whole process.

*Ruffus* is a way of automating the plumbing in your pipeline: You supply the python functions which perform the data transformation, and tell *Ruffus* how these pipeline `task` functions are connected up. *Ruffus* will make sure that the right data flows down your pipeline in the right way at the right time.

---

**Note:** *Ruffus* refers to each stage of your pipeline as a *task*.

---

### 1.3.2 Importing *Ruffus*

The most convenient way to use *Ruffus* is to import the various names directly:

```
from ruffus import *
```

This will allow *Ruffus* terms to be used directly in your code. This is also the style we have adopted for this manual.

**If any of these clash with names in your code, you can use qualified names instead:**

```
import ruffus

ruffus.pipeline_printout("...")
```

*Ruffus* uses only standard python syntax.

There is no need to install anything extra or to have your script “preprocessed” to run your pipeline.

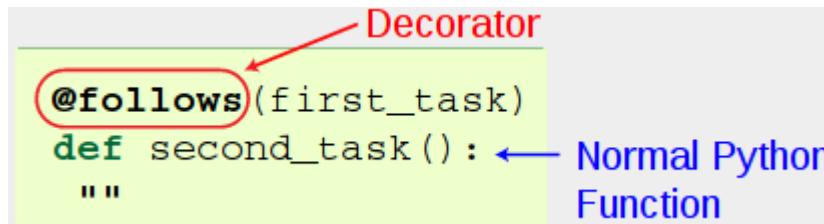
### 1.3.3 *Ruffus* decorators

To let *Ruffus* know that which python functions are part of your pipeline, they need to be tagged or annotated using *Ruffus* decorators.

Decorators have been part of the Python language since version 2.4. Common examples from the standard library include `@staticmethod` and `classmethod`.

`decorators` start with a @ prefix, and take a number of parameters in parenthesis, much like in a function call.

`decorators` are placed before a normal python function.



Multiple decorators can be stacked as necessary in whichever order:

```
@follows(first_task)
@follows(another_task)
@originate(range(5))
def second_task():
    """
```

*Ruffus* decorators do not otherwise alter the underlying function. These can still be called normally.

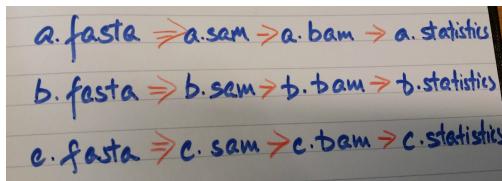
## 1.3.4 Your first *Ruffus* pipeline

### 1. Write down the file names

*Ruffus* is designed for data moving through a computational pipeline as a series of files.

It is also possible to use *Ruffus* pipelines without using intermediate data files but for your first efforts, it is probably best not to subvert its canonical design.

The first thing when designing a new *Ruffus* pipeline is to sketch out the set of file names for the pipeline on paper:



Here we have a number of DNA sequence files (`*.fasta`)

1. mapped to a genome (`*.sam`), and
2. compressed (`*.bam`) before being
3. summarised statistically (`*.statistics`)

The first striking thing is that all of the files following the same **consistent naming scheme**.

---

**Note:** The most important part of a Ruffus pipeline is to have a consistent naming scheme for your files. This allows you to build sane pipelines.

---

In this case, each of the files at the same stage share the same file extension, e.g. `(.sam)`. This is usually the simplest and most sensible choice. (We shall see in later chapters that *Ruffus* supports more complicated naming patterns so long as they are consistent.)

### 2. Write the python functions for each stage

Next, we can sketch out the python functions which do the actual work for the pipeline.

---

**Note:**

1. These are normal python functions with the important proviso that
  - (a) The first parameter contains the **Input** (file names)
  - (b) The second parameter contains the **Output** (file names)

You can otherwise supply as many parameters as is required.
2. Each python function should only take a *Single Input* at a time

All the parallelism in your pipeline should be handled by *Ruffus*. Make sure each function analyses one thing at a time.

---

*Ruffus* refers to a pipelined function as a *task*.

The code for our three task functions look something like:

```
#  
#      STAGE 1 fasta->sam  
#  
def map_dna_sequence(input_file,  
                      output_file):  
    """  
        Sketch of real mapping function  
        We can do the mapping ourselves  
        or call some other programme:  
        os.system("stampy %s %s..." % (input_file, output_file))  
    """  
    ii = open(input_file)  
    oo = open(output_file, "w")  
  
    #  
    #      STAGE 2 sam->bam  
    #  
def compress_sam_file(input_file,  
                      output_file):  
    """  
        Sketch of real compression function  
    """  
    ii = open(input_file)  
    oo = open(output_file, "w")  
  
    #  
    #      STAGE 3 bam->statistics  
    #  
def summarise_bam_file(input_file,  
                      output_file,  
                      extra_stats_parameter):  
    """  
        Sketch of real analysis function  
    """  
    ii = open(input_file)  
    oo = open(output_file, "w")
```

If we were calling our functions manually, without the benefit of *Ruffus*, we would need the following sequence of calls:

```
# STAGE 1  
map_dna_sequence("a.fasta", "a.sam")  
map_dna_sequence("b.fasta", "b.sam")  
map_dna_sequence("c.fasta", "c.sam")  
  
# STAGE 2  
compress_sam_file("a.sam", "a.bam")  
compress_sam_file("b.sam", "b.bam")  
compress_sam_file("c.sam", "c.bam")  
  
# STAGE 3  
summarise_bam_file("a.bam", "a.statistics")  
summarise_bam_file("b.bam", "b.statistics")  
summarise_bam_file("c.bam", "c.statistics")
```

### 3. Link the python functions into a pipeline

*Ruffus* makes exactly the same function calls on your behalf. However, first, we need to tell *Ruffus* what the arguments should be for each of the function calls.

- The **Input** is easy: This is either the starting file set (`*.fasta`) or whatever is produced by the previous stage.
- The **Output** file name is the same as the **Input** but with the appropriate extension.

These are specified using the *Ruffus* `@transform` decorator as follows:

```
from ruffus import *

starting_files = ["a.fasta", "b.fasta", "c.fasta"]

#
#    STAGE 1 fasta->sam
#
@transform(starting_files,
           suffix(".fasta"),
           ".sam")                                # Input = starting files
                                                # suffix = .fasta
                                                # Output suffix = .sam

def map_dna_sequence(input_file,
                      output_file):
    ii = open(input_file)
    oo = open(output_file, "w")

#
#    STAGE 2 sam->bam
#
@transform(map_dna_sequence,
           suffix(".sam"),
           ".bam")                                # Input = previous stage
                                                # suffix = .sam
                                                # Output suffix = .bam

def compress_sam_file(input_file,
                      output_file):
    ii = open(input_file)
    oo = open(output_file, "w")

#
#    STAGE 3 bam->statistics
#
@transform(map_dna_sequence,
           suffix(".bam"),
           ".statistics",
           "use_linear_model")                    # Input = previous stage
                                                # suffix = .bam
                                                # Output suffix = .statistics
                                                # Extra statistics parameter

def summarise_bam_file(input_file,
                        output_file,
                        extra_stats_parameter):
    """
    Sketch of real analysis function
    """
    ii = open(input_file)
    oo = open(output_file, "w")
```

### 4. `@transform` syntax

1. The 1st parameter for `@transform` is the **Input**.

This is either the set of starting data or the name of the previous pipeline function.

*Ruffus* chains together the stages of a pipeline by linking the **Output** of the previous stage into the **Input** of the next.

2. The 2nd parameter is the current *suffix*  
(i.e. our **Input** file extensions of ".fasta" or ".sam" or ".bam")
3. The 3rd parameter is what we want our **Output** file name to be after *suffix* string substitution (e.g. .fasta - > .sam).  
This works because we are using a sane naming scheme for our data files.
4. Other parameters can be passed to @transform and they will be forwarded to our python pipeline function.

The functions that do the actual work of each stage of the pipeline remain unchanged. The role of *Ruffus* is to make sure each is called in the right order, with the right parameters, running in parallel (using multiprocessing if desired).

## 5. Run the pipeline!

---

### Note: Key Ruffus Terminology:

A **task** is an annotated python function which represents a recipe or stage of your pipeline.

A **job** is each time your recipe is applied to a piece of data, i.e. each time *Ruffus* calls your function.

Each **task** or pipeline recipe can thus have many **jobs** each of which can work in parallel on different data.

---

Now we can run the pipeline with the *Ruffus* function *pipeline\_run*:

```
pipeline_run()
```

This produces three sets of results in parallel, as you might expect:

```
>>> pipeline_run()
Job  = [a.fasta -> a.sam] completed
Job  = [b.fasta -> b.sam] completed
Job  = [c.fasta -> c.sam] completed
Completed Task = map_dna_sequence
Job  = [a.sam -> a.bam] completed
Job  = [b.sam -> b.bam] completed
Job  = [c.sam -> c.bam] completed
Completed Task = compress_sam_file
Job  = [a.bam -> a.statistics, use_linear_model] completed
Job  = [b.bam -> b.statistics, use_linear_model] completed
Job  = [c.bam -> c.statistics, use_linear_model] completed
Completed Task = summarise_bam_file
```

To work out which functions to call, *pipeline\_run* finds the **last task** function of your pipeline, then works out all the other functions this depends on, working backwards up the chain of dependencies automatically.

We can specify this end point of your pipeline explicitly:

```
>>> pipeline_run(target_tasks = [summarise_bam_file])
```

This allows us to only run part of the pipeline, for example:

```
>>> pipeline_run(target_tasks = [compress_sam_file])
```

---

**Note:** The *example code* can be copied and pasted into a python command shell.

---

## 1.4 Chapter 2: Transforming data in a pipeline with `@transform`

See also:

- [Manual Table of Contents](#)
- [@transform syntax](#)

**Note:** Remember to look at the example code:

- [Chapter 1: Python Code for Transforming data in a pipeline with @transform](#)

### 1.4.1 Review



Computational pipelines transform your data in stages until the final result is produced. Ruffus automates the plumbing in your pipeline. You supply the python functions which perform the data transformation, and tell Ruffus how these pipeline stages or *task* functions are connected together.

**Note:** The best way to design a pipeline is to:

- write down the file names of the data as it flows across your pipeline
- write down the names of functions which transforms the data at each stage of the pipeline.

### 1.4.2 Task functions as recipes

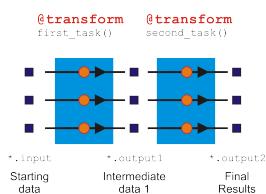
Each *task* function of the pipeline is a recipe or `rule` which can be applied repeatedly to our data.

For example, one can have

- a `compile()` *task* which will compile any number of source code files, or
- a `count_lines()` *task* which will count the number of lines in any file or
- an `align_dna()` *task* which will align the DNA of many chromosomes.

### 1.4.3 `@transform` is a 1 to 1 operation

`@transform` is a 1:1 operation because for each input, it generates one output.



This is obvious when you count the number of jobs at each step. In our example pipeline, there are always three jobs moving through in step at each stage (*task*).

Each **Input** or **Output** is not limited, however, to a single filename. Each job can accept, for example, a pair of files as its **Input**, or generate more than one file or a dictionary or numbers as its **Output**.

When each job outputs a pair of files, this does not generate two jobs downstream. It just means that the successive *task* in the pipeline will receive a list or tuple of files as its input parameter.

**Note:** The different sort of decorators in Ruffus determine the *topology* of your pipeline, i.e. how the jobs from different tasks are linked together seamlessly.

`@transform` always generates one **Output** for one **Input**.

In the later parts of the tutorial, we will encounter more decorators which can *split up*, or *join together* or *group* inputs.

In other words, using other decorators **Input** and **Output** can have **many to one**, **many to many** etc. relationships.

---

### A pair of files as the Input

Let us rewrite our previous example so that the **Input** of the first task are matching pairs of DNA sequence files, processed in tandem.

```
from ruffus import *

starting_files = [("a.1.fastq", "a.2.fastq"),
                  ("a.1.fastq", "a.2.fastq"),
                  ("a.1.fastq", "a.2.fastq")]

#
#      STAGE 1  fasta->sam
#
@transform(starting_files,                                # Input = starting files
           suffix(".1.fastq"),                         # suffix = .1.fastq
           ".sam")                                     # Output suffix = .sam

def map_dna_sequence(input_files,
                      output_file):
    # remember there are two input files now
    i1l = open(input_files[0])
    i1r = open(input_files[1])
    oo = open(output_file, "w")
```

The only changes are to the first task:

```
pipeline_run()
Job   = [[a.1.fastq, a.2.fastq] -> a.sam] completed
Job   = [[a.1.fastq, a.2.fastq] -> a.sam] completed
Job   = [[a.1.fastq, a.2.fastq] -> a.sam] completed
Completed Task = map_dna_sequence
```

`suffix` always matches only the first file name in each **Input**.

#### 1.4.4 Input and Output parameters

**Ruffus** chains together different tasks by taking the **Output** from one job and plugging it automatically as the **Input** of the next.

The first two parameters of each job are the **Input** and **Output** parameters respectively.

In the above example, we have:

```
>>> pipeline_run()
Job   = [a.bam -> a.statistics, use_linear_model] completed
Job   = [b.bam -> b.statistics, use_linear_model] completed
Job   = [c.bam -> c.statistics, use_linear_model] completed
Completed Task = summarise_bam_file
```

Table 1.1: Parameters for summarise\_bam\_file()

Inputs	Outputs	Extra
"a.bam"	"a.statistics"	"use_linear_model"
"b.bam"	"b.statistics"	"use_linear_model"
"c.bam"	"c.statistics"	"use_linear_model"

**Extra** parameters are for the consumption of `summarise_bam_file()` and will not be passed to the next task.

Ruffus was designed for pipelines which save intermediate data in files. This is not compulsory but saving your data in files at each step provides many advantages:

1. Ruffus can use file system time stamps to check if your pipeline is up to date
2. Your data is persistent across runs
3. This is a good way to pass large amounts of data across processes and computational nodes

Nevertheless, *all* the `task` parameters can include anything which suits your workflow, from lists of files, to numbers, sets or tuples. *Ruffus* imposes few constraints on what *you* would like to send to each stage of your pipeline.

*Ruffus* does, however, assume that if the **Input** and **Output** parameter contains strings, these will be interpreted as file names required by and produced by that job. As we shall see, the modification times of these file names indicate whether that part of the pipeline is up to date or needs to be rerun.

## 1.5 Chapter 3: More on @transform-ing data

See also:

- *Manual Table of Contents*
- `@transform` syntax

---

**Note:** Remember to look at the example code:

- *Chapter 3: Python Code for More on @transform-ing data*
- 

### 1.5.1 Review



Computational pipelines transform your data in stages until the final result is produced. *Ruffus* automates the plumbing in your pipeline. You supply the python functions which perform the data transformation, and tell *Ruffus* how these pipeline stages or `task` functions are connected together.

---

**Note: The best way to design a pipeline is to:**

- write down the file names of the data as it flows across your pipeline

- write down the names of functions which transforms the data at each stage of the pipeline.
- 

*Chapter 1: An introduction to basic Ruffus syntax* described the bare bones of a simple *Ruffus* pipeline.

Using the *Ruffus* `@transform` decorator, we were able to specify the data files moving through our pipeline so that our specified task functions could be invoked.

This may seem like a lot of effort and complication for something so simple: a couple of simple python function calls we could have invoked ourselves. However, By letting *Ruffus* manage your pipeline parameters, you will get the following features for free:

1. Only out-of-date parts of the pipeline will be re-run
2. Multiple jobs can be run in parallel (on different processors if possible)
3. Pipeline stages can be chained together automatically. This means you can apply your pipeline just as easily to 1000 files as to 3.

## 1.5.2 Running pipelines in parallel

Even though three sets of files have been specified for our initial pipeline, and they can be processed completely independently, by default *Ruffus* runs each of them serially in succession.

To ask *Ruffus* to run them in parallel, all you have to do is to add a `multiprocess` parameter to `pipeline_run`:

```
>>> pipeline_run(multiprocess = 5)
```

In this case, we are telling *Ruffus* to run a maximum of 5 jobs at the same time. Since we only have three sets of data, that is as much parallelism as we are going to get...

## 1.5.3 Up-to-date jobs are not re-run unnecessarily

A job will be run only if the output file timestamps are out of date. If you ran our example code a second time, nothing would happen because all the work is already complete.

We can check the details by asking *Ruffus* for more verbose output

```
>>> pipeline_run(verbose = 4)
Task = map_dna_sequence
    All jobs up to date
Task = compress_sam_file
    All jobs up to date
Task = summarise_bam_file
    All jobs up to date
```

**Nothing happens because:**

- `a.sam` was created later than `a.1.fastq` and `a.2.fastq`, and
- `a.bam` was created later than `a.sam` and
- `a.statistics` was created later than `a.bam`.

and so on...

**Let us see what happens if we recreated the file `a.1.fastq` so that it appears as if 1 out of the original data files is out of**

```
open("a.1.fastq", "w")
pipeline_run(multiprocess = 5)
```

The up to date jobs are cleverly ignored and only the out of date files are reprocessed.

```
>>> open("a.1.fastq", "w")
>>> pipeline_run(verbose=2)
Job   = [[b.1.fastq, b.2.fastq] -> b.sam] # unnecessary: already up to date
Job   = [[c.1.fastq, c.2.fastq] -> c.sam] # unnecessary: already up to date
Job   = [[a.1.fastq, a.2.fastq] -> a.sam] completed
Completed Task = map_dna_sequence
Job   = [b.sam -> b.bam] # unnecessary: already up to date
Job   = [c.sam -> c.bam] # unnecessary: already up to date
Job   = [a.sam -> a.bam] completed
Completed Task = compress_sam_file
Job   = [b.bam -> b.statistics, use_linear_model] # unnecessary: already up to date
Job   = [c.bam -> c.statistics, use_linear_model] # unnecessary: already up to date
Job   = [a.bam -> a.statistics, use_linear_model] completed
Completed Task = summarise_bam_file
```

#### 1.5.4 Defining pipeline tasks out of order

The examples so far assumes that all your pipelined tasks are defined in order. (`first_task` before `second_task`). This is usually the most sensible way to arrange your code.

If you wish to refer to tasks which are not yet defined, you can do so by quoting the function name as a string and wrapping it with the *indicator class* `output_from(...)` so that *Ruffus* knows this is a *task name*, not a file name

```
#-----
#
#   second task
#
#   task name string wrapped in output_from(...)
@transform(output_from("first_task"), suffix(".output.1"), ".output2")
def second_task(input_files, output_file):
    with open(output_file, "w"): pass

#-----
#
#   first task
#
@transform(first_task_params, suffix(".start"),
          [".output.1",
           ".output.extra.1"],
          "some_extra.string.for_example", 14)
def first_task(input_files, output_file_pair,
               extra_parameter_str, extra_parameter_num):
    for output_file in output_file_pair:
        with open(output_file, "w"):
            pass

#-----
#
#       Run
```

```
#  
pipeline_run([second_task])
```

You can also refer to tasks (functions) in other modules, in which case the full qualified name must be used:

```
@transform(output_from("other_module.first_task"), suffix(".output.1"), ".output2")  
def second_task(input_files, output_file):  
    pass
```

### 1.5.5 Multiple dependencies

Each task can depend on more than one antecedent simply by chaining to a list in `@transform`

```
#  
# third_task depends on both first_task() and second_task()  
#  
@transform([first_task, second_task], suffix(".output.1"), ".output2")  
def third_task(input_files, output_file):  
    with open(output_file, "w"): pass
```

`third_task()` depends on and follows both `first_task()` and `second_task()`. However, these latter two tasks are independent of each other and can and will run in parallel. This can be clearly shown for our example if we added a little randomness to the run time of each job:

```
time.sleep(random.random())
```

The execution of `first_task()` and `second_task()` jobs will be interleaved and they finish in no particular order:

```
>>> pipeline_run([third_task], multiprocess = 6)
Job   = [[job3.a.start, job3.b.start] -> [job3.a.output.1, job3.a.output.extra.1], some_e
Job   = [[job6.a.start, job6.b.start] -> [job6.a.output.1, job6.a.output.extra.1], some_e
Job   = [[job1.a.start, job1.b.start] -> [job1.a.output.1, job1.a.output.extra.1], some_e
Job   = [[job4.a.start, job4.b.start] -> [job4.a.output.1, job4.a.output.extra.1], some_e
Job   = [[job5.a.start, job5.b.start] -> [job5.a.output.1, job5.a.output.extra.1], some_e
Completed Task = second_task
Job   = [[job2.a.start, job2.b.start] -> [job2.a.output.1, job2.a.output.extra.1], some_e
```

---

**Note:** See the *example code*

---

### 1.5.6 `@follows`

If there is some extrinsic reason one non-dependent task has to precede the other, then this can be specified explicitly using `@follows`:

```
#  
#   @follows specifies a preceding task  
#  
@follows("first_task")  
@transform(second_task_params, suffix(".start"),  
        [".output.1",  
         ".output.extra.1"],  
        "some_extra.string.for_example", 14)
```

```
def second_task(input_files, output_file_pair,
                extra_parameter_str, extra_parameter_num):
```

`@follows` specifies either a preceding task (e.g. `first_task`), or if it has not yet been defined, the name (as a string) of a task function (e.g. "`first_task`").

With the addition of `@follows`, all the jobs of `second_task()` start *after* those from `first_task()` have finished:

```
>>> pipeline_run([third_task], multiprocess = 6)
Job  = [[job2.a.start, job2.b.start] -> [job2.a.output.1, job2.a.output.extra.1], some_e
Job  = [[job3.a.start, job3.b.start] -> [job3.a.output.1, job3.a.output.extra.1], some_e
Job  = [[job1.a.start, job1.b.start] -> [job1.a.output.1, job1.a.output.extra.1], some_e
Completed Task = first_task
Job  = [[job4.a.start, job4.b.start] -> [job4.a.output.1, job4.a.output.extra.1], some_e
Job  = [[job6.a.start, job6.b.start] -> [job6.a.output.1, job6.a.output.extra.1], some_e
Job  = [[job5.a.start, job5.b.start] -> [job5.a.output.1, job5.a.output.extra.1], some_e
Completed Task = second_task
```

### 1.5.7 Making directories automatically with `@follows` and `mkdir`

`@follows` is also useful for making sure one or more destination directories exist before a task is run.

*Ruffus* provides special syntax to support this, using the special `mkdir` indicator class. For example:

```
#           @follows specifies both a preceding task and a directory name
#
@follows("first_task", mkdir("output/results/here"))
@transform(second_task_params, suffix(".start"),
          [".output.1",
           ".output.extra.1"],
          "some_extra.string.for_example", 14)
def second_task(input_files, output_file_pair,
                extra_parameter_str, extra_parameter_num):
```

Before `second_task()` is run, the `output/results/here` directory will be created if necessary.

### 1.5.8 Globs in the Input parameter

- As a syntactic convenience, *Ruffus* also allows you to specify a `glob` pattern (e.g. `*.txt`) in the **Input** parameter.
- `glob` patterns will be automatically specify all matching file names as the **Input**.
- Any strings within **Input** which contain the letters: `*?[]` will be treated as a `glob` pattern.

The first function in our initial *Ruffus* pipeline example could have been written as:

```
#           STAGE 1 fasta->sam
#
@transform("*.fasta",                                     # Input = glob
          suffix(".fasta"),                                # suffix = .fasta
          ".sam")                                         # Output suffix = .sam
def map_dna_sequence(input_file,
                     output_file):
    """
```

### 1.5.9 Mixing Tasks and Globs in the Input parameter

*glob* patterns, references to tasks and file names strings can be mixed freely in (nested) python lists and tuples in the **Input** parameter.

For example, a task function can chain to the **Output** from multiple upstream tasks:

```
@transform([task1, task2,
            "aa*.fasta",
            "zz.fasta"]
           suffix(".fasta"),
           ".sam")
def map_dna_sequence(input_file,
                      output_file):
    """
```

In all cases, *Ruffus* tries to do the right thing, and to make the simple or obvious case require the simplest, least onerous syntax.

If sometimes *Ruffus* does not behave the way you expect, please write to the authors: it may be a bug!

*Chapter 5: Understanding how your pipeline works with pipeline\_printout(...)* and *Chapter 6: Running Ruffus from the command line with ruffus.cmdline* will show you how to make sure that your intentions are reflected in *Ruffus* code.

## 1.6 Chapter 4: Creating files with @originate

See also:

- *Manual Table of Contents*
- *@originate syntax in detail*

---

**Note:** Remember to look at the example code:

- *Chapter 4: Python Code for Creating files with @originate*
- 

### 1.6.1 Simplifying our example with @originate

Our previous pipeline example started off with a set of files which we had to create first.

This is a common task: pipelines have to start *somewhere*.

Ideally, though, we would only want to create these starting files if they didn't already exist. In other words, we want a sort of `@transform` which makes files from nothing (`None?`).

This is exactly what `@originate` helps you to do.

Rewriting our pipeline with `@originate` gives the following three steps:

```
from ruffus import *
#-----
#  create initial files
#
@originate([
    ['job1.a.start', 'job1.b.start'],
    ['job2.a.start', 'job2.b.start'],
    ['job3.a.start', 'job3.b.start']] )
```

```
def create_initial_file_pairs(output_files):
    # create both files as necessary
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

#-----
#   first task
@transform(create_initial_file_pairs, suffix(".start"), ".output.1")
def first_task(input_files, output_file):
    with open(output_file, "w"): pass

#-----
#   second task
@transform(first_task, suffix(".output.1"), ".output.2")
def second_task(input_files, output_file):
    with open(output_file, "w"): pass

#
#           Run
#
pipeline_run([second_task])

Job = [None -> [job1.a.start, job1.b.start]] completed
Job = [None -> [job2.a.start, job2.b.start]] completed
Job = [None -> [job3.a.start, job3.b.start]] completed
Completed Task = create_initial_file_pairs
    Job = [[job1.a.start, job1.b.start] -> job1.a.output.1] completed
    Job = [[job2.a.start, job2.b.start] -> job2.a.output.1] completed
    Job = [[job3.a.start, job3.b.start] -> job3.a.output.1] completed
Completed Task = first_task
    Job = [job1.a.output.1 -> job1.a.output.2] completed
    Job = [job2.a.output.1 -> job2.a.output.2] completed
    Job = [job3.a.output.1 -> job3.a.output.2] completed
Completed Task = second_task
```

## 1.7 Chapter 5: Understanding how your pipeline works with *pipeline\_printout(...)*

See also:

- *Manual Table of Contents*
- *pipeline\_printout(...)* syntax
- *Python Code for this chapter*

---

Note:

- Whether you are learning or developing ruffus pipelines, your best friend is *pipeline\_printout(...)* This shows the exact parameters and files as they are passed through the pipeline.
- We also strongly recommend you use the Ruffus.cmdline convenience module which will take care of all the command line arguments for you. See *Chapter 6: Running Ruffus from the command line with ruffus.cmdline*.

### 1.7.1 Printing out which jobs will be run

`pipeline_printout(...)` takes the same parameters as `pipeline_run` but just prints the tasks which are and are not up-to-date.

The `verbose` parameter controls how much detail is displayed.

Let us take the pipelined code we previously wrote in [Chapter 3 More on @transform-ing data and @originate](#) but call `pipeline_printout(...)` instead of `pipeline_run(...)`. This lists the tasks which will be run in the pipeline:

```
>>> import sys  
>>> pipeline_printout(sys.stdout, [second_task])
```

---

```
Tasks which will be run:
```

```
Task = create_initial_file_pairs  
Task = first_task  
Task = second_task
```

---

To see the input and output parameters of each job in the pipeline, try increasing the verbosity from the default (1) to 3 (See `code`)

This is very useful for checking that the input and output parameters have been specified correctly.

### 1.7.2 Determining which jobs are out-of-date or not

It is often useful to see which tasks are or are not up-to-date. For example, if we were to run the pipeline in full, and then modify one of the intermediate files, the pipeline would be partially out of date.

Let us start by run the pipeline in full but then modify `job1.a.output.1` so that the second task appears out-of-date:

```
pipeline_run([second_task])  
  
# "touch" job1.stage1  
open("job1.a.output.1", "w").close()
```

Run `pipeline_printout(...)` with a verbosity of 5.

This will tell you exactly why `second_task(...)` needs to be re-run: because `job1.a.output.1` has a file modification time *after* `job1.a.output.2` (highlighted):

```
>>> pipeline_printout(sys.stdout, [second_task], verbose = 5)  
  
Tasks which will be run:  
  
Task = second_task  
    Job = [job1.a.output.1  
          -> job1.a.output.2]  
  
>>> # File modification times shown for out of date files  
        Job needs update:  
        Input files:  
            * 05 Dec 2013 12:04:52.80: job1.a.output.1  
        Output files:  
            * 05 Dec 2013 12:01:29.01: job1.a.output.2
```

```
Job   = [ job2.a.output.1
         -> job2.a.output.2]
Job up-to-date
Job   = [ job3.a.output.1
         -> job3.a.output.2]
Job up-to-date
```

---

N.B. At a verbosity of 5, even jobs which are up-to-date will be displayed.

### 1.7.3 Getting a list of all tasks in a pipeline

If you just wanted a list of all tasks (Ruffus decorated function names), then you can just run `Run_pipeline_get_task_names(...)`.

This doesn't touch any pipeline code or even check to see if the pipeline is connected up properly.

However, it is sometimes useful to allow users at the command line to choose from a list of possible tasks as a target.

## 1.8 Chapter 6: Running *Ruffus* from the command line with `ruffus.cmdline`

### See also:

- *Manual table of Contents*

We find that much of our *Ruffus* pipeline code is built on the same template and this is generally a good place to start developing a new pipeline.

From version 2.4, *Ruffus* includes an optional `Ruffus.cmdline` module that provides support for a set of common command line arguments. This makes writing *Ruffus* pipelines much more pleasant.

### 1.8.1 Template for argparse

All you need to do is copy these 6 lines

```
from ruffus import *

parser = cmdline.get_argparse(description='WHAT DOES THIS PIPELINE DO?')

# <<<---- add your own command line options like --input_file here
# parser.add_argument("--input_file")

options = parser.parse_args()

# standard python logger which can be synchronised across concurrent Ruffus tasks
logger, logger_mutex = cmdline.setup_logging (__name__, options.log_file, options.verbose)

# <<<---- pipelined functions go here

cmdline.run (options)
```

You are recommended to use the standard `argparse` module but the deprecated `optparse` module works as well. (See *below* for the template)

## 1.8.2 Command Line Arguments

Ruffus.`cmdline` by default provides these predefined options:

```
-v, --verbose
--version
-L, --log_file

# tasks
-T, --target_tasks
--forced_tasks
-j, --jobs
--use_threads

# printout
-n, --just_print

# flow chart
--flowchart
--key_legend_in_graph
--draw_graph_horizontally
--flowchart_format

# check sum
--touch_files_only
--checksum_file_name
--recreate_database
```

## 1.8.3 1) Logging

The script provides for logging both to the command line:

```
myscript -v
myscript --verbose
```

and an optional log file:

```
# keep tabs on yourself
myscript --log_file /var/log/secret.logbook
```

Logging is ignored if neither `--verbose` or `--log_file` are specified on the command line

Ruffus.`cmdline` automatically allows you to write to a shared log file via a proxy from multiple processes. However, you do need to use `logging_mutex` for the log files to be synchronised properly across different jobs:

```
with logging_mutex:

    logger_proxy.info("Look Ma. No hands")
```

Logging is set up so that you can write

**A) Only to the log file:**

```
logger.info("A message")
```

**B) Only to the display:**

```
logger.debug("A message")
```

**C) To both simultaneously:**

```
from ruffus cmdline import MESSAGE  
  
logger.log(MESSAGE, "A message")
```

### 1.8.4 2) Tracing pipeline progress

This is extremely useful for understanding what is happening with your pipeline, what tasks and which jobs are up-to-date etc.

See *Chapter 5: Understanding how your pipeline works with pipeline\_printout(...)*

To trace the pipeline, call script with the following options

```
# well-mannered, reserved  
myscript --just_print  
myscript -n  
  
or  
  
# extremely loquacious  
myscript --just_print --verbose 5  
myscript -n -v5
```

Increasing levels of verbosity (--verbose to --verbose 5) provide more detailed output

### 1.8.5 3) Printing a flowchart

This is the subject of *Chapter 7: Displaying the pipeline visually with pipeline\_printout\_graph(...)*.

Flowcharts can be specified using the following option:

```
myscript --flowchart xxxchart.svg
```

The extension of the flowchart file indicates what format the flowchart should take, for example, `svg`, `jpg` etc.

Override with `--flowchart_format`

### 1.8.6 4) Running in parallel on multiple processors

Optionally specify the number of parallel strands of execution and which is the last *target* task to run. The pipeline will run starting from any out-of-date tasks which precede the *target* and proceed no further beyond the *target*.

```
myscript --jobs 15 --target_tasks "final_task"
myscript -j 15
```

### 1.8.7 5) Setup checkpointing so that *Ruffus* knows which files are out of date

The *checkpoint file* uses the value set in the environment (DEFAULT\_RUFFUS\_HISTORY\_FILE).

If this is not set, it will default to .ruffus\_history.sqlite in the current working directory.

Either can be changed on the command line:

```
myscript --checksum_file_name mychecksum.sqlite
```

#### Recreating checkpoints

Create or update the checkpoint file so that all existing files in completed jobs appear up to date

Will stop sensibly if current state is incomplete or inconsistent

```
myscript --recreate_database
```

#### Touch files

As far as possible, create empty files with the correct timestamp to make the pipeline appear up to date.

```
myscript --touch_files_only
```

### 1.8.8 6) Skipping specified options

Note that particular options can be skipped (not added to the command line), if they conflict with your own options, for example:

```
# see below for how to use get_argparse
parser = cmdline.get_argparse( description='WHAT DOES THIS PIPELINE DO?',
                               # Exclude the following options: --log_file --key_legend_in_
                               ignored_args = ["log_file", "key_legend_in_graph"])
```

### 1.8.9 7) Displaying the version

Note that the version for your script will default to "% (prog) s 1.0" unless specified:

```
parser = cmdline.get_argparse( description='WHAT DOES THIS PIPELINE DO?',
                               version = "my_programme.py v. 2.23")
```

### 1.8.10 Template for optparse

deprecated since python 2.7

```
#  
#   Using optparse (new in python v 2.6)  
#  
from ruffus import *  
  
parser = cmdline.get_optgparse(version="%prog 1.0", usage = "\n\n      %prog [options]")  
  
#   <<<---- add your own command line options like --input_file here  
# parser.add_option("-i", "--input_file", dest="input_file", help="Input file")  
  
(options, remaining_args) = parser.parse_args()  
  
# logger which can be passed to ruffus tasks  
logger, logger_mutex = cmdline.setup_logging ("this_program", options.log_file, options.verb  
  
#   <<<---- pipelined functions go here  
  
cmdline.run (options)
```

## 1.9 Chapter 7: Displaying the pipeline visually with *pipeline\_printout\_graph(...)*

See also:

- *Manual Table of Contents*
- *pipeline\_printout\_graph(...)* syntax

---

**Note:** Remember to look at the example code:

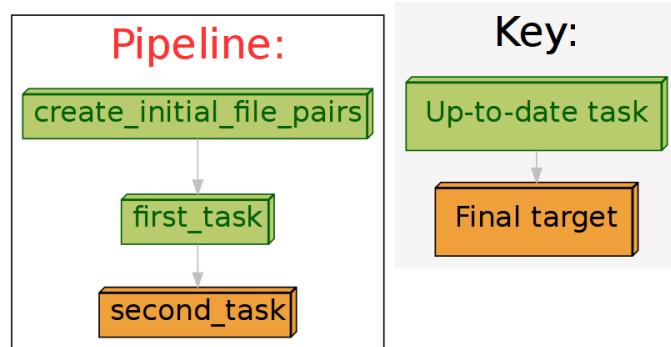
- *Chapter 7: Python Code for Displaying the pipeline visually with pipeline\_printout\_graph(...)*
- 

### 1.9.1 Printing out a flowchart of our pipeline

It is all very well being able to trace the data flow through the pipeline as text. Sometimes, however, we need a bit of eye-candy!

We can see a flowchart for our fledgling pipeline by executing:

```
pipeline_printout_graph (    'flowchart.svg',  
                            'svg',  
                            [second_task],  
                            no_key_legend = False)
```

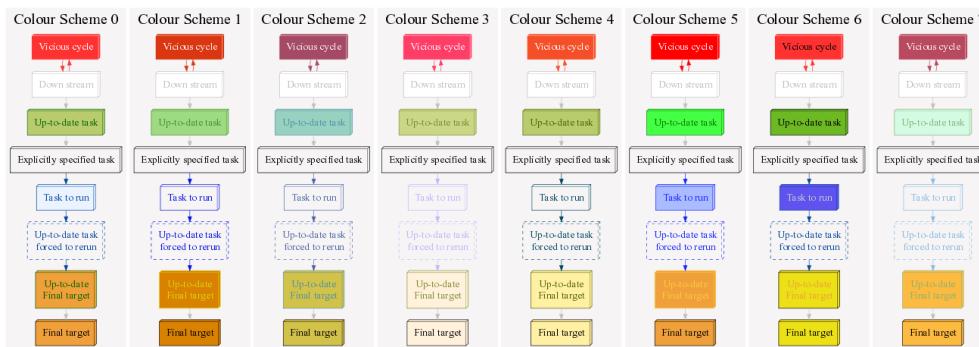


Flowcharts can be printed in a large number of formats including jpg, svg, png and pdf.

**Note:** Flowcharts rely on the dot programme from Graphviz.

Please make sure this is installed.

There are 8 standard colour schemes, but you can further customise all the colours to your satisfaction:



See [here](#) for example code.

## 1.9.2 Command line options made easier with ruffus.cmdline

If you are using `ruffus.cmdline`, then you can easily ask for a flowchart from the command line:

```
your_script.py --flowchart pipeline_flow_chart.png
```

The output format is deduced from the extension but can be specified manually:

```
# specify format. Otherwise, deduced from the extension
your_script.py --flowchart pipeline_flow_chart.png --flowchart_format png
```

Print the flow chart horizontally or vertically...

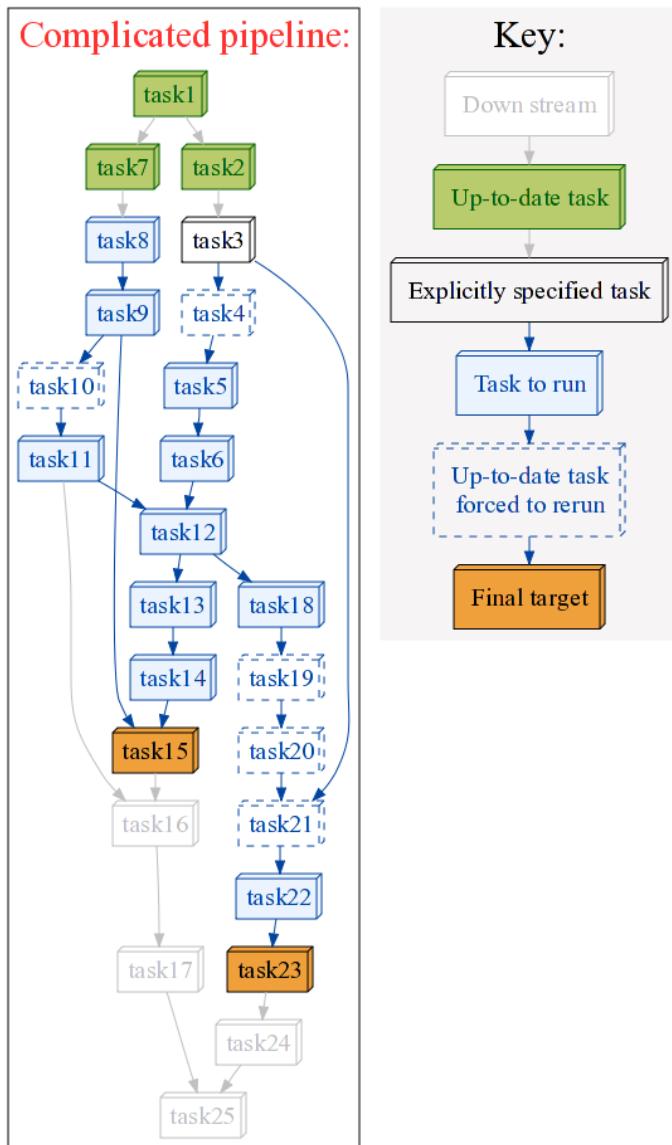
```
# flowchart proceeds from left to right , rather than from top to bottom
your_script.py --flowchart pipeline_flow_chart.png --draw_graph_horizontally
```

...with or without a key legend

```
# Draw key legend
your_script.py --flowchart pipeline_flow_chart.png --key_legend_in_graph
```

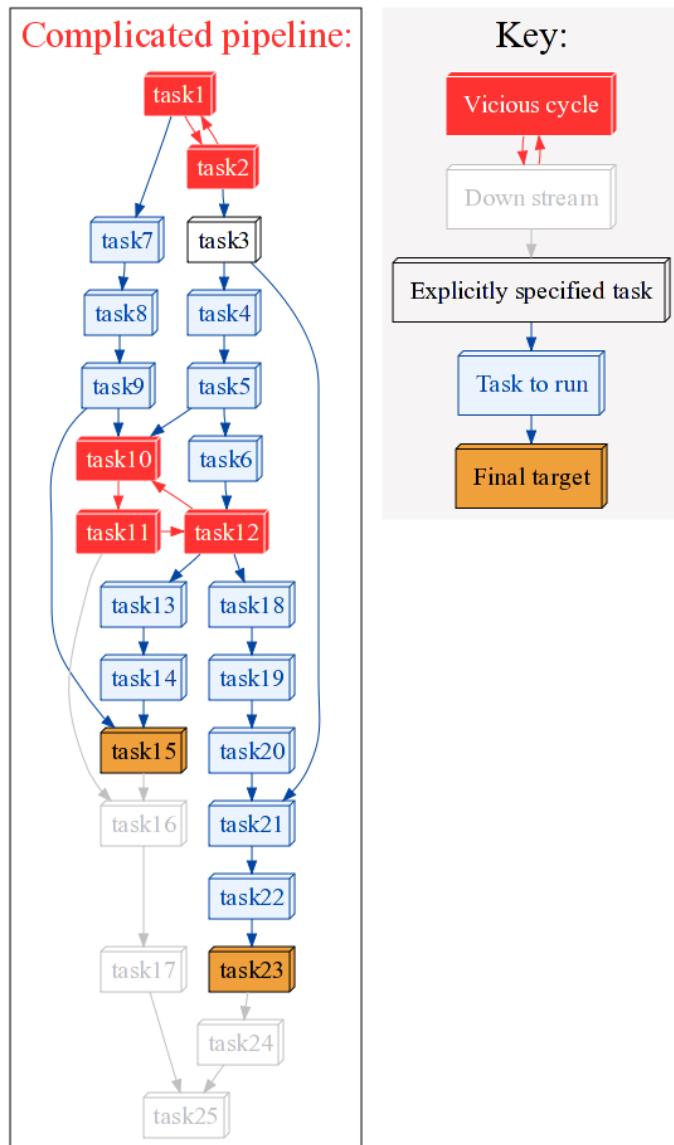
### 1.9.3 Horribly complicated pipelines!

Flowcharts are especially useful if you have really complicated pipelines, such as



### 1.9.4 Circular dependency errors in pipelines!

Especially, if the pipeline is not set up properly, and vicious circular dependencies are present:



## 1.10 Chapter 8: Specifying output file names with `formatter()` and `regex()`

See also:

- *Manual Table of Contents*
- `suffix()` syntax
- `formatter()` syntax
- `regex()` syntax

---

**Note:** Remember to look at the example code:

- *Chapter 8: Python Code for Specifying output file names with formatter() and regex()*

### 1.10.1 Review



Computational pipelines transform your data in stages until the final result is produced. The most straightforward way to use Ruffus is to hold the intermediate results after each stage in a series of files with related file names.

Part of telling Ruffus how these pipeline stages or *task* functions are connected together is to write simple rules for how the file names for each stage follow on from each other. Ruffus helps you to specify these file naming rules.

**Note:** The best way to design a pipeline is to:

- Write down the file names of the data as it flows across your pipeline. Do these file names follow a pattern ?
- Write down the names of functions which transforms the data at each stage of the pipeline.

### 1.10.2 A different file name *suffix()* for each pipeline stage

The easiest and cleanest way to write Ruffus pipelines is to use a different suffix for each stage of your pipeline.

We used this approach in *Chapter 1: An introduction to basic Ruffus syntax* and in code from *Chapter 3: More on @transform-ing data*:

#Task Name:	File suffices
create_initial_file_pairs	*.start
first_task	*.output.1
second_task	*.output.2

There is a long standing convention of using file suffices to denote file type: For example, a “compile” task might convert **source** files of type `*.c` to **object** files of type `*.o`.

We can think of Ruffus tasks comprising :

- recipes in `@transform(...)` for transforming file names: changing `.c` to `a.o` (e.g. `AA.c -> AA.o BB.c -> BB.o`)
- recipes in a task function `def foo_bar()` for transforming your data: from **source** `.c` to **object** `.o`

Let us review the Ruffus syntax for doing this:

```

@transform( create_initial_file_pairs,   # Input:  Name of previous task(s)
           suffix(".start"),
           ".output.1")                  # Matching suffix
           # Replacement string

def first_task(input_files, output_file):
    with open(output_file, "w"): pass
  
```

#### 1. Input:

The first parameter for `@transform` can be a mixture of one or more:

- previous tasks (e.g. `create_initial_file_pairs`)
- file names (all python strings are treated as paths)
- glob specifications (e.g `*.c`, `/my/path/*.foo`)

Each element provides an input for the task. So if the previous task `create_initial_file_pairs` has five outputs, the next `@transform` task will accept these as five separate inputs leading to five independent jobs.

## 2. `suffix()`:

The second parameter `suffix(".start")` must match the end of the first string in each input. For example, `create_initial_file_pairs` produces the list `['job1.a.start', 'job1.b.start']`, then `suffix(".start")` must matches the first string, i.e. `'job1.a.start'`. If the input is nested structure, this would be iterated through recursively to find the first string.

---

**Note:** Inputs which do not match the suffix are discarded altogether.

---

## 3. Replacement:

The third parameter is the replacement for the suffix. The pair of input strings in the step3 example produces the following output parameter

```
input_parameters = ['job1.a.start', 'job1.b.start']
matching_input   = 'job1.a.start'
output_parameter = 'job1.a.output.1'
```

When the pipeline is run, this results in the following equivalent call to `first_task(...)`:

```
first_task(['job1.a.start', 'job1.b.start'], 'job1.a.output.1');
```

The replacement parameter can itself be a list or any arbitrary complicated structure:

```
@transform(create_initial_file_pairs,                      # Input
           suffix(".a.start"),                         # Matching suffix
           [".output.a.1", ".output.b.1", 45])        # Replacement list
def first_task(input_files, output_parameters):
    print "input_parameters = ", input_files
    print "output_parameters = ", output_parameters
```

In which case, all the strings are used as replacements, other values are left untouched, and we obtain the following:

```
# job #1
input  = ['job1.a.start',     'job1.b.start']
output = ['job1.output.a.1',  'job1.output.b.1', 45]

# job #2
input  = ['job2.a.start',     'job2.b.start']
output = ['job2.output.a.1',  'job2.output.b.1', 45]

# job #3
input  = ['job3.a.start',     'job3.b.start']
output = ['job3.output.a.1',  'job3.output.b.1', 45]
```

Note how task function is called with the value 45 *verbatim* because it is not a string.

### 1.10.3 *formatter()* manipulates pathnames and regular expression

*suffix()* replacement is the cleanest and easiest way to generate suitable output file names for each stage in a pipeline. Often, however, we require more complicated manipulations to specify our file names. For example,

- It is common to have to change directories from a *data* directory to a *working* directory as the first step of a pipeline.
- Data management can be simplified by separate files from each pipeline stage into their own directory.
- Information may have to be decoded from data file names, e.g. "experiment373.IBM.03March2002.txt"

Though *formatter()* is much more powerful, the principle and syntax are the same: we take string elements from the **Input** and perform some replacements to generate the **Output** parameters.

#### *formatter()*

- Allows easy manipulation of path subcomponents in the style of `os.path.split()`, and `os.path.basename`
- Uses familiar python `string.format` syntax (See [string.format examples](#). )
- Supports optional regular expression (`re`) matches including named captures.
- Can refer to any file path (i.e. python string) in each input and is not limited like *suffix()* to the first string.
- Can even refer to individual letters within a match

#### Path name components

*formatter()* breaks down each input pathname into path name components which can then be recombined in whichever way by the replacement string.

Given an example string of :

```
input_string = "/directory/to/a/file.name.ext"
formatter()
```

the path components are:

- basename: The base name *excluding* extension, "file.name"
- ext : The *extension*, ".ext"
- path : The *dirname*, "/directory/to/a"
- subdir : A list of sub-directories in the path in reverse order, ["a", "to", "directory", "/"]
- subpath : A list of descending sub-paths in reverse order, ["/directory/to/a", "/directory/to", "/directory", "/"]

The replacement string refers to these components by using python `string.format` style curly braces. "`{NAME}`"

We refer to an element from the Nth input string by index, for example:

- "`{ext[0]}`" is the extension of the first file name string in **Input**.
- "`{basename[1]}`" is the basename of the second file name in **Input**.

- "`{basename[1][0:3]}`" are the first three letters from the basename of the second file name in **Input**.

`subdir`, `subpath` were designed to help you navigate directory hierachies with the minimum of fuss. For example, you might want to graft a hierachical path to another location: "`{subpath[0][2]}/from/{subdir[0][0]}/{basename[0]}`" neatly replaces just one directory ("to") in the path with another ("from"):

```
replacement_string = "{subpath[0][2]}/from/{subdir[0][0]}/{basename[0]}"

input_string      = "/directory/to/a/file.name.ext"
result_string    = "/directory/from/a/file.name.ext"
```

## Filter and parse using regular expressions

Regular expression matches can be used with the similar syntax. Our example string can be parsed using the following regular expression:

```
input_string = "/directory/to/a/file.name.ext"
formatter(r"/directory/(.+)/(?P<MYFILENAME>)\.ext")
```

We capture part of the path using `(.+)`, and the base name using `(?P<MYFILENAME>)`. These matching subgroups can be referred to by index but for greater clarity the second named capture can also be referred to by name, i.e. `{MYFILENAME}`.

The regular expression components for the first string can thus be referred to as follows:

- `{0[0]}` : The entire match captured by index, `/directory/to/a/file.name.ext`
- `{1[0]}` : The first match captured by index, `to/a`
- `{2[0]}` : The second match captured by index, `file.name`
- `{MYFILENAME[0]}` : The match captured by name, `file.name`

If each input consists of a list of paths such as `['job1.a.start', 'job1.b.start', 'job1.c.start']`, we can match each of them separately by using as many regular expressions as necessary. For example:

```
input_string = ['job1.a.start', 'job1.b.start', 'job1.c.start']
# Regular expression matches for 1st, 2nd but not 3rd element
formatter(".+a.start", "b.start$")
```

Or if you only wanted regular expression matches for the second file name (string), pad with None:

```
input_string = ['job1.a.start', 'job1.b.start', 'job1.c.start']
# Regular expression matches for 2nd but not 1st or 3rd elements
formatter(None, "b.start$")
```

## Using `@transform()` with `formatter()`

We can put these together in the following example:

```
from ruffus import *

# create initial files
@originate([
    ['job1.a.start', 'job1.b.start'],
    ['job2.a.start', 'job2.b.start'],
    ['job3.a.start', 'job3.c.start']
])
```

```

def create_initial_file_pairs(output_files):
    # create both files as necessary
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

#-----
#
#   formatter
#

#   first task
@transform(create_initial_file_pairs,                                # Input
          formatter(".+/job(?P<JOBNUMBER>\d+).a.start",
                    ".+/job[123].b.start"),
          ["{path[0]}/jobs{JOBNUMBER[0]}.output.a.1",
           "{path[1]}/jobs{JOBNUMBER[0]}.output.b.1", 45])           # Extract job number
# Match only "b" files
# Replacement list

def first_task(input_files, output_parameters):
    print "input_parameters = ", input_files
    print "output_parameters = ", output_parameters

#
#       Run
#
pipeline_run(verbose=0)

```

This produces:

```

input_parameters = ['job1.a.start',
                   'job1.b.start']
output_parameters = ['/home/lg/src/temp/jobs1.output.a.1',
                     '/home/lg/src/temp/jobs1.output.b.1', 45]

input_parameters = ['job2.a.start',
                   'job2.b.start']
output_parameters = ['/home/lg/src/temp/jobs2.output.a.1',
                     '/home/lg/src/temp/jobs2.output.b.1', 45]

```

Notice that job3 has 'job3.c.start' as the second file. This fails to match the regular expression and is discarded.

---

**Note:** Failed regular expression mismatches are ignored.

*formatter()* regular expressions are thus very useful in filtering out all files which do not match your specified criteria.

If your some of your task inputs have a mixture of different file types, a simple `Formatter(".txt$")`, for example, will make your code a lot simpler...

---

## string substitution for “extra” arguments

The first two arguments for Ruffus task functions are special because they are the **Input** and **Output** parameters which link different stages of a pipeline.

Python strings in these arguments are names of data files whose modification times indicate whether the pipeline is up to date or not.

Other arguments to task functions are not passed down the pipeline but consumed. Any python strings they contain do not need to be file names. These extra arguments are very useful for passing data to pipelined tasks, such as shared values, loggers, programme options etc.

One helpful feature is that strings in these extra arguments are also subject to *formatter()* string substitution. This means you can leverage the parsing capabilities of Ruffus to decode any information about the pipeline data files. These might include the directories you are running in and parts of the file name.

For example, if we would want to know which files go with which “job number” in the previous example:

```
from ruffus import *

#    create initial files
@originate([
    ['job1.a.start', 'job1.b.start'],
    ['job2.a.start', 'job2.b.start'],
    ['job3.a.start', 'job3.c.start']    ])
def create_initial_file_pairs(output_files):
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

#-----
#    print job number as an extra argument
#
#    first task
@transform(create_initial_file_pairs,                                # Input
          formatter(".+/job(?P<JOBNUMBER>\d+).a.start",
                    ".+/job[123].b.start"),
          [{"path[0]}/jobs{JOBNUMBER[0]}.output.a.1",
           "{path[1]}/jobs{JOBNUMBER[0]}.output.b.1"],                  # Replacement list
          "{JOBNUMBER[0]}")
def first_task(input_files, output_parameters, job_number):
    print job_number, ":", input_files

pipeline_run(verbose=0)

>>> pipeline_run(verbose=0)
1 : ['job1.a.start', 'job1.b.start']
2 : ['job2.a.start', 'job2.b.start']
```

### Changing directories using *formatter()* in a zo...

Here is a more fun example. We would like to feed the denizens of a zoo. Unfortunately, the file names for these are spread over several directories. Ideally, we would like their food supply to be grouped more sensibly. And, of course, we only want to feed the animals, not the plants.

I have colour coded the input and output files for this task to show how we would like to rearrange them:

```

crocodile/reptiles.wild.animals  →→→ reptiles/wild.crocodile.food
dog/mammals.tame.animals          →→→ mammals/tame.dog.food
dog/mammals.wild.animals          →→→ mammals/wild.dog.food
lion/mammals.handreared.animals   →→→ mammals/handreared.lion.food
lion/mammals.wild.animals          →→→ mammals/wild.lion.food
lion/mammals.wild.animals          →→→ mammals/wild.lion.food
Roses are not animals!
rose/flowering.handreared.plants  ➔➔➔

from ruffus import *

# Make directories
@mkdir(["tiger", "lion", "dog", "crocodile", "rose"])

@originate(
    # List of animals and plants
    [
        "tiger/mammals.wild.animals",
        "lion/mammals.wild.animals",
        "lion/mammals.handreared.animals",
        "dog/mammals.tame.animals",
        "dog/mammals.wild.animals",
        "crocodile/reptiles.wild.animals",
        "rose/flowering.handreared.plants"])
def create_initial_files(output_file):
    with open(output_file, "w") as oo: pass

    # Put different animals in different directories depending on their clade
    @transform(create_initial_files,
              # Input
              formatter(".+/ (?P<clade>\w+). (?P<tame>\w+).animals"),
              # Only animals: ignore plants
              "{subpath[0][1]}/{clade[0]}/{tame[0]}.{subdir[0][0]}.food",
              # Replacement
              "{subpath[0][1]}/{clade[0]}",
              "{subdir[0][0]}",
              "{tame[0]}")

    def feed(input_file, output_file, new_directory, animal_name, tameness):
        print "Food for the {tameness:11s} {animal_name:9s} = {output_file:90s} will be placed in {new_directory:90s}"
        pipeline_run(verbose=0)

```

We can see that the food for each animal are now grouped by clade in the same directory, which makes a lot more sense...

Note how we used `subpath[0][1]` to move down one level of the file path to build a new file name.

```

>>> pipeline_run(verbose=0)
Food for the wild      crocodile = ./reptiles/wild.crocodile.food will be placed in ./reptiles/wild
Food for the tame      dog       = ./mammals/tame.dog.food      will be placed in ./mammals/tame
Food for the wild      dog       = ./mammals/wild.dog.food     will be placed in ./mammals/wild
Food for the handreared lion     = ./mammals/handreared.lion.food will be placed in ./mammals/handreared
Food for the wild      lion     = ./mammals/wild.lion.food      will be placed in ./mammals/wild
Food for the wild      tiger    = ./mammals/wild.tiger.food     will be placed in ./mammals/wild

```

#### 1.10.4 `regex()` manipulates via regular expressions

If you are a hard core regular expressions fan, you may want to use `regex()` instead of `suffix()` or `formatter()`.

---

**Note:** `regex()` uses regular expressions like `formatter()` but

- It only matches the first file name in the input. As described above, `formatter()` can match any one or more of the input filename strings.
- It does not understand file paths so you may have to perform your own directory / file name parsing.
- String replacement uses syntax borrowed from `re.sub()`, rather than building a result from parsed regular expression (and file path) components

In general `formatter()` is more powerful and was introduced from version 2.4 is intended to be a more user friendly replacement for `regex()`.

---

Let us see how the previous zoo example looks with `regex()`:

`formatter()` code:

```
# Put different animals in different directories depending on their clade
@transform(create_initial_files,                                     # Input
          formatter(".+/ (?P<clade>\w+). (?P<tame>\w+).animals"),
          "{subpath[0][1]}/{clade[0]}/{tame[0]}.{subdir[0][0]}.food",      # Replacement
          "{subpath[0][1]}/{clade[0]}",
          "{subdir[0][0]}",
          "{tame[0]}")
def feed(input_file, output_file, new_directory, animal_name, tameness):
    print "Food for the {tameness:11s} {animal_name:9s} = {output_file:90s} will be placed in"

# new_directory
# animal_name
# tameness
```

`regex()` code:

```
# Put different animals in different directories depending on their clade
@transform(create_initial_files,                                     # Input
          regex(r"(.*?/?)(\w+)/ (?P<clade>\w+). (?P<tame>\w+).animals"),
          r"\1/\g<clade>/\g<tame>. \2.food",                                # Replacement
          r"\1/\g<clade>",
          r"\2",
          r"\g<tame>")
def feed(input_file, output_file, new_directory, animal_name, tameness):
    print "Food for the {tameness:11s} {animal_name:9s} = {output_file:90s} will be placed in"

# new_directory
# animal_name
# tameness
```

The regular expression to parse the input file path safely was a bit hairy to write, and it is not clear that it handles all edge conditions (e.g. files in the root directory). Apart from that, if the limitations of `regex()` do not preclude its use, then the two approaches are not so different in practice.

## 1.11 Chapter 9: Preparing directories for output with `@mkdir()`

See also:

- *Manual Table of Contents*
- *@follows(mkdir()) syntax in detail*
- *@mkdir syntax in detail*

---

**Note:** Remember to look at the example code:

- Chapter 9: Python Code for Preparing directories for output with @mkdir()

### 1.11.1 Overview

In **Chapter 3**, we saw that we could use `@follows(mkdir())` to ensure that output directories exist:

```
#  
#     create_new_files() @follows mkdir  
#  
@follows(mkdir("output/results/here"))  
@originate(["output/results/here/a.start_file",  
           "output/results/here/b.start_file"])  
def create_new_files(output_file_pair):  
    pass
```

This ensures that the decorated task follows (`@follows`) the making of the specified directory (`mkdir()`).

Sometimes, however, the **Output** is intended not for any single directory but a group of destinations depending on the parsed contents of **Input** paths.

### 1.11.2 Creating directories after string substitution in a zoo...

You may remember *this example* from **Chapter 8**:

We want to feed the denizens of a zoo. The original file names are spread over several directories and we group their food supply by the *clade* of the animal in the following manner:

```
crocodile/reptiles.wild.animals      → reptiles/wild.crocodile.food  
dog/mammals.tame.animals             → mammals/tame.dog.food  
dog/mammals.wild.animals             → mammals/wild.dog.food  
lion/mammals.handreared.animals     → mammals/handreared.lion.food  
lion/mammals.wild.animals            → mammals/wild.lion.food  
lion/mammals.wild.animals            → mammals/wild.lion.food  
Roses are not animals!  
rose/flowering.handreared.plants     ✖→  
  
# Put different animals in different directories depending on their clade  
@transform(create_initial_files,          # Input  
  
          formatter(".+/(?P<clade>\w+).(?P<tame>\w+).animals"),        # Only animals: ignore  
          " {subpath[0][1]}/{clade[0]}/{tame[0]}.food",                      # Replacement  
          " {subpath[0][1]}/{clade[0]}",                                         # new_directory  
          " {subdir[0][0]}",                                                 # animal_name  
          " {tame[0]}")                                                       # tameness  
def feed(input_file, output_file, new_directory, animal_name, tameness):  
    print "%40s -> %90s" % (input_file, output_file)  
    # this blows up  
    # open(output_file, "w")
```

The example code from **Chapter 8** is, however, incomplete. If we were to actually create the specified files we would realise that we had forgotten to create the destination directories `reptiles`, `mammals` first!

#### using `formatter()`

We could of course create directories manually. However, apart from being tedious and error prone, we have already gone to some lengths to parse out the directories for `@transform`. Why don't we use the same logic to make the directories?

Can you see the parallels between the syntax for `@mkdir` and `@transform`?

```
# create directories for each clade
@mkdir(    create_initial_files,                                     # Input

          formatter(".+/(?P<clade>\w+).(?P<tame>\w+).animals"),
          "{subpath[0][1]}/{clade[0]}")                                     # Only animals: ignore
                                                               # new_directory

# Put animals of each clade in the same directory
@transform(create_initial_files,                                     # Input

          formatter(".+/(?P<clade>\w+).(?P<tame>\w+).animals"),
          "{subpath[0][1]}/{clade[0]}/{tame[0]}.{subdir[0][0]}.food", # Replacement
                                                               # new_directory
          "{subpath[0][1]}/{clade[0]}",
          "{subdir[0][0]}",
          "{tame[0]}")                                                 # animal_name
                                                               # tameness

def feed(input_file, output_file, new_directory, animal_name, tameness):
    print "%40s -> %90s" % (input_file, output_file)
    # this works now
    open(output_file, "w")
```

See the *example code*

### using `regex()`

If you are particularly fond of using regular expression to parse file paths, you could also use `regex()`:

```
# create directories for each clade
@mkdir(    create_initial_files,                                     # Input

          regex(r"(.*)/?(\w+)/ (?P<clade>\w+).(?P<tame>\w+).animals"),
          r"\1/\g<clade>")                                     # Only animals: ignore
                                                               # new_directory

# Put animals of each clade in the same directory
@transform(create_initial_files,                                     # Input

          formatter(".+/(?P<clade>\w+).(?P<tame>\w+).animals"),
          "{subpath[0][1]}/{clade[0]}/{tame[0]}.{subdir[0][0]}.food", # Replacement
                                                               # new_directory
          "{subpath[0][1]}/{clade[0]}",
          "{subdir[0][0]}",
          "{tame[0]}")                                                 # animal_name
                                                               # tameness

def feed(input_file, output_file, new_directory, animal_name, tameness):
    print "%40s -> %90s" % (input_file, output_file)
    # this works now
    open(output_file, "w")
```

## 1.12 Chapter 10: Checkpointing: Interrupted Pipelines and Exceptions

See also:

- *Manual Table of Contents*

---

**Note:** Remember to look at the example code:

- *Chapter 10: Python Code for Checkpointing: Interrupted Pipelines and Exceptions*
- 

### 1.12.1 Overview



Computational pipelines transform your data in stages until the final result is produced.

By default, *Ruffus* uses file modification times for the **input** and **output** to determine whether each stage of a pipeline is up-to-date or not. But what happens when the task function is interrupted, whether from the command line or by error, half way through writing the output?

In this case, the half-formed, truncated and corrupt **Output** file will look newer than its **Input** and hence up-to-date.

### 1.12.2 Interrupting tasks

Let us try with an example:

```

from ruffus import *
import sys, time

#   create initial files
@originate(['job1.start'])
def create_initial_files(output_file):
    with open(output_file, "w") as oo: pass

#-----
#   long task to interrupt
#
@transform(create_initial_files, suffix(".start"), ".output")
def long_task(input_files, output_file):
    with open(output_file, "w") as ff:
        ff.write("Unfinished...")
        # sleep for 2 seconds here so you can interrupt me
        sys.stderr.write("Job started. Press ^C to interrupt me now...\n")
        time.sleep(2)
        ff.write("\nFinished")
        sys.stderr.write("Job completed.\n")

#
#       Run
pipeline_run([long_task])
  
```

When this script runs, it pauses in the middle with this message:

Job started. Press ^C to interrupt me now...

If you interrupted the script by pressing Control-C at this point, you will see that `job1.output` contains only `Unfinished....`. However, if you should rerun the interrupted pipeline again, Ruffus ignores the corrupt, incomplete file:

```
>>> pipeline_run([long_task])
Job started. Press ^C to interrupt me now...
Job completed
```

And if you had run `pipeline_printout`:

```
>>> pipeline_printout(sys.stdout, [long_task], verbose=3)
=====
Tasks which will be run:

Task = long_task
    Job  = [job1.start
            -> job1.output]
    # Job needs update: Previous incomplete run leftover: [job1.output]
```

We can see that *Ruffus* magically knows that the previous run was incomplete, and that `job1.output` is detritus that needs to be discarded.

### 1.12.3 Checkpointing: only log completed jobs

All is revealed if you were to look in the working directory. *Ruffus* has created a file called `.ruffus_history.sqlite`. In this [SQLite](#) database, *Ruffus* logs only those files which are the result of a completed job, all other files are suspect. This file checkpoint database is a fail-safe, not a substitute for checking file modification times. If the **Input** or **Output** files are modified, the pipeline will rerun.

By default, *Ruffus* saves only file timestamps to the SQLite database but you can also add a checksum of the pipeline task function body or parameters. This behaviour can be controlled by setting the `checksum_level` parameter in `pipeline_run()`. For example, if you do not want to save any timestamps or checksums:

```
pipeline_run(checksum_level = 0)

CHECKSUM_FILE_TIMESTAMPS      = 0      # only rerun when the file timestamps are out of date
CHECKSUM_HISTORY_TIMESTAMPS   = 1      # Default: also rerun when the history shows a job as
CHECKSUM_FUNCTIONS            = 2      # also rerun when function body has changed
CHECKSUM_FUNCTIONS_AND_PARAMS = 3      # also rerun when function parameters or function body
```

---

**Note:** Checksums are calculated from the [pickled](#) string for the function code and parameters. If pickling fails, Ruffus will degrade gracefully to saving just the timestamp in the SQLite database.

---

### 1.12.4 Do not share the same checkpoint file across for multiple pipelines!

The name of the Ruffus python script is not saved in the checkpoint file along side timestamps and checksums. That means that you can rename your pipeline source code file without having to rerun the pipeline! The tradeoff is that if multiple pipelines are run from the same directory, and save their histories to the same SQLite database file, and if their file names overlap (all of these are bad ideas anyway!), this is bound to be a source of confusion.

Luckily, the name and path of the checkpoint file can be also changed for each pipeline

### 1.12.5 Setting checkpoint file names

**Warning:** Some file systems do not appear to support SQLite at all:  
 There are reports that SQLite databases have [file locking problems](#) on Lustre.  
 The best solution would be to keep the SQLite database on an alternate compatible file system away from the working directory if possible.

#### environment variable `DEFAULT_RUFFUS_HISTORY_FILE`

The name of the checkpoint file is the value of the environment variable `DEFAULT_RUFFUS_HISTORY_FILE`.

```
export DEFAULT_RUFFUS_HISTORY_FILE=/some/where/.ruffus_history.sqlite
```

This gives considerable flexibility, and allows a system-wide policy to be set so that all Ruffus checkpoint files are set logically to particular paths.

---

**Note:** It is your responsibility to make sure that the requisite destination directories for the checkpoint files exist beforehand!

Where this is missing, the checkpoint file defaults to `.ruffus_history.sqlite` in your working directory

#### Setting the checkpoint file name manually

This checkpoint file name can always be overridden as a parameter to Ruffus functions:

```
pipeline_run(history_file = "XXX")
pipeline_printout(history_file = "XXX")
pipeline_printout_graph(history_file = "XXX")
```

There is also built in support in `Ruffus cmdline`. So if you use this module, you can simply add to your command line:

```
# use a custom checkpoint file
myscript --checksum_file_name .myscript.ruffus_history.sqlite
```

This takes precedence over everything else.

### 1.12.6 Useful checkpoint file name policies `DEFAULT_RUFFUS_HISTORY_FILE`

If the pipeline script is called `test/bin/scripts/run.me.py`, then these are the resulting checkpoint files locations:

#### Example 1: same directory, different name

If the environment variable is:

```
export DEFAULT_RUFFUS_HISTORY_FILE=.{basename}.ruffus_history.sqlite
```

Then the job checkpoint database for `run.me.py` will be `.run.me.ruffus_history.sqlite`

```
/test/bin/scripts/run.me.py  
/common/path/for/job_history/scripts/.run.me.ruffus_history.sqlite
```

#### Example 2: Different directory, same name

```
export DEFAULT_RUFFUS_HISTORY_FILE=/common/path/for/job_history/.{basename}.ruffus_history.sqlite  
  
/common/path/for/job_history/.run.me.ruffus_history.sqlite
```

#### Example 2: Different directory, same name but keep one level of subdirectory to disambiguate

```
export DEFAULT_RUFFUS_HISTORY_FILE=/common/path/for/job_history/{subdir[0]}/{basename}.ruffus_history.sqlite  
  
/common/path/for/job_history/scripts/.run.me.ruffus_history.sqlite
```

#### Example 2: nested in common directory

```
export DEFAULT_RUFFUS_HISTORY_FILE=/common/path/for/job_history/{path}/{basename}.ruffus_history.sqlite  
  
/common/path/for/job_history/test/bin/scripts/.run.me.ruffus_history.sqlite
```

### 1.12.7 Regenerating the checkpoint file

Occasionally you may need to re-generate the checkpoint file.

This could be necessary:

- because you are upgrading from a previous version of Ruffus without checkpoint file support
- on the rare occasions when the SQLite file becomes corrupted and has to deleted
- if you wish to circumvent the file checking of Ruffus after making some manual changes!

To do this, it is only necessary to call `pipeline_run` appropriately:

```
CHECKSUM_REGENERATE = 2  
pipeline(touch_files_only = CHECKSUM_REGENERATE)
```

Similarly, if you are using `Ruffus.cmdline`, you can call:

```
myscript --recreate_database
```

Note that this regenerates the checkpoint file to reflect the existing *Input*, *Output* files on disk. In other words, the onus is on you to make sure there are no half-formed, corrupt files. On the other hand, the pipeline does not need to have been previously run successfully for this to work. Essentially, Ruffus, pretends to run the pipeline, while logging all the files with consistent file modification times, stopping at the first tasks which appear out of date or incomplete.

### 1.12.8 Rules for determining if files are up to date

The following simple rules are used by *Ruffus*.

1. The pipeline stage will be rerun if:

- If any of the **Input** files are new (newer than the **Output** files)
  - If any of the **Output** files are missing
2. In addition, it is possible to run jobs which create files from scratch.
    - If no **Input** file names are supplied, the job will only run if any *output* file is missing.
  3. Finally, if no **Output** file names are supplied, the job will always run.

### 1.12.9 Missing files generate exceptions

If the *inputs* files for a job are missing, the task function will have no way to produce its *output*. In this case, a `MissingInputFileError` exception will be raised automatically. For example,

```
task.MissingInputFileError: No way to run job: Input file ['a.1'] does not exist  
for Job = ["a.1" -> "a.2", "A file"]
```

### 1.12.10 Caveats: Coarse Timestamp resolution

Note that modification times have precision to the nearest second under some older file systems (ext2/ext3?). This may be also be true for networked file systems.

*Ruffus* supplements the file system time resolution by independently recording the timestamp at full OS resolution (usually to at least the millisecond) at job completion, when presumably the **Output** files will have been created.

However, *Ruffus* only does this if the discrepancy between file time and system time is less than a second (due to poor file system timestamp resolution). If there are large mismatches between the two, due for example to network time slippage, misconfiguration etc, *Ruffus* reverts to using the file system time and adds a one second delay between jobs (via `time.sleep()`) to make sure input and output file stamps are different.

If you know that your filesystem has coarse-grained timestamp resolution, you can always revert to this very conservative behaviour, at the prices of some annoying 1s pauses, by setting `pipeline_run(one_second_per_job = True)`

### 1.12.11 Flag files: Checkpointing for the paranoid

One other way of checkpointing your pipelines is to create an extra “flag” file as an additional **Output** file name. The flag file is only created or updated when everything else in the job has completed successfully and written to disk. A missing or out of date flag file then would be a sign for Ruffus that the task never completed properly in the first place.

This used to be much the best way of performing checkpointing in Ruffus and is still the most bulletproof way of proceeding. For example, even the loss or corruption of the checkpoint file, would not affect things greatly.

Nevertheless flag files are largely superfluous in modern *Ruffus*.

## 1.13 Chapter 11: Pipeline topologies and a compendium of *Ruffus* decorators

See also:

- *Manual Table of Contents*
- *decorators*

### 1.13.1 Overview

Computational pipelines transform your data in stages until the final result is produced.

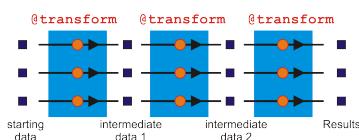
You can visualise your pipeline data flowing like water down a system of pipes. *Ruffus* has many ways of joining up your pipes to create different topologies.

**Note:** The best way to design a pipeline is to:

- Write down the file names of the data as it flows across your pipeline.
  - Draw lines between the file names to show how they should be connected together.
- 

### 1.13.2 @transform

So far, our data files have been flowing through our pipelines independently in lockstep.

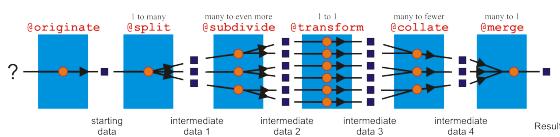


If we drew a graph of the data files moving through the pipeline, all of our flowcharts would look like something like this.

The `@transform` decorator connects up your data files in 1 to 1 operations, ensuring that for every **Input**, a corresponding **Output** is generated, ready to go into the next pipeline stage. If we start with three sets of starting data, we would end up with three final sets of results.

### 1.13.3 A bestiary of *Ruffus* decorators

Very often, we would like to transform our data in more complex ways, this is where other *Ruffus* decorators come in.



### 1.13.4 @originate

- Introduced in **Chapter 3 More on @transform-ing data and @originate**, `@originate` generates **Output** files from scratch without the benefits of any **Input** files.

### 1.13.5 @merge

- A **many to one** operator.
- The last decorator at the far right to the figure, `@merge` merges multiple **Input** into one **Output**.

### 1.13.6 @split

- A **one to many** operator,
- `@split` is the evil twin of `@merge`. It takes a single set of **Input** and splits them into multiple smaller pieces.
- The best part of `@split` is that we don't necessarily have to decide ahead of time *how many* smaller pieces it should produce. If we have encounter a larger file, we might need to split it up into more fragments for greater parallelism.
- Since `@split` is a **one to many** operator, if you pass it **many** inputs (e.g. via `@transform`, it performs an implicit `@merge` step to make one set of **Input** that you can redistribute into a different number of pieces. If you are looking to split *each* **Input** into further smaller fragments, then you need `@subdivide`

### 1.13.7 @subdivide

- A **many to even more** operator.
- It takes each of multiple **Input**, and further subdivides them.
- Uses `suffix()`, `formatter()` or `regex()` to generate **Output** names from its **Input** files but like `@split`, we don't have to decide ahead of time *how many* smaller pieces each **Input** should be further divided into. For example, a large **Input** files might be subdivided into 7 pieces while the next job might, however, split its **Input** into just 4 pieces.

### 1.13.8 @collate

- A **many to fewer** operator.
- `@collate` is the opposite twin of `subdivide`: it takes multiple **Output** and groups or collates them into bundles of **Output**.
- `@collate` uses `formatter()` or `regex()` to generate **Output** names.
- All **Input** files which map to the same **Output** are grouped together into one job (one task function call) which produces one **Output**.

### 1.13.9 Combinatorics

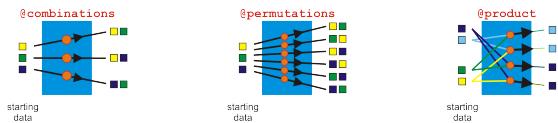
More rarely, we need to generate a set of **Output** based on a combination or permutation or product of the **Input**.

For example, in bioinformatics, we might need to look for all instances of a set of genes in the genomes of a different number of species. In other words, we need to find the `@product` of XXX genes x YYY species.

*Ruffus* provides decorators modelled on the “Combinatorial generators” in the Standard Python `itertools` library.

To use combinatoric decorators, you need to explicitly include them from *Ruffus*:

```
import ruffus
from ruffus import *
from ruffus.combinatorics import *
```



### 1.13.10 `@product`

- Given several sets of **Input**, it generates all versus all **Output**. For example, if there are four sets of **Input** files, `@product` will generate `WWW x XXX x YYY x ZZZ` **Output**.
- Uses *formatter* to generate unique **Output** names from components parsed from *any* parts of *any* specified files in all **Input** sets. In the above example, this allows the generation of `WWW x XXX x YYY x ZZZ` unique names.

### 1.13.11 `@combinations`

- Given one set of **Input**, it generates the combinations of r-length tuples among them.
- Uses *formatter* to generate unique **Output** names from components parsed from *any* parts of *any* specified files in all **Input** sets.
- For example, given **Input** called A, B and C, it will generate: A-B, A-C, B-C
- The order of **Input** items is ignored so either A-B or B-A will be included, not both
- Self-vs-self combinations (A-A) are excluded.

### 1.13.12 `@combinations_with_replacement`

- Given one set of **Input**, it generates the combinations of r-length tuples among them but includes self-vs-self combinations.
- Uses *formatter* to generate unique **Output** names from components parsed from *any* parts of *any* specified files in all **Input** sets.
- For example, given **Input** called A, B and C, it will generate: A-A, A-B, A-C, B-B, B-C, C-C

### 1.13.13 `@permutations`

- Given one set of **Input**, it generates the permutations of r-length tuples among them. This excludes self-vs-self combinations but includes all orderings (A-B and B-A).
- Uses *formatter* to generate unique **Output** names from components parsed from *any* parts of *any* specified files in all **Input** sets.
- For example, given **Input** called A, B and C, it will generate: A-A, A-B, A-C, B-A, B-C, C-A, C-B

## 1.14 Chapter 12: Splitting up large tasks / files with `@split`

See also:

- Manual Table of Contents*
- `@split` syntax

- Example code for this chapter

### 1.14.1 Overview

A common requirement in computational pipelines is to split up a large task into small jobs which can be run on different processors, (or sent to a computational cluster). Very often, the number of jobs depends dynamically on the size of the task, and cannot be known beforehand.

*Ruffus* uses the `@split` decorator to indicate that the `task` function will produce an indeterminate number of independent *Outputs* from a single *Input*.

### 1.14.2 Example: Calculate variance for a large list of numbers in parallel

Suppose we wanted to calculate the `variance` for 100,000 numbers, how can we parallelise the calculation so that we can get an answer as speedily as possible?

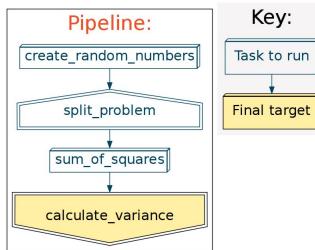
We need to

- break down the problem into manageable chunks
- solve these in parallel, possibly on a computational cluster and then
- merge the partial solutions back together for a final result.

To complicate things, we usually do not want to hard-code the number of parallel chunks beforehand. The degree of parallelism is often only apparent as we process our data.

**Ruffus** was designed to solve such problems which are common, for example, in bioinformatics and genomics.

A flowchart for our variance problem might look like this:



(In this toy example, we create our own starting data in `create_random_numbers()`.)

### 1.14.3 Output files for `@split`

The *Ruffus* decorator `@split` is designed specifically with this run-time flexibility in mind:

```

@split(create_random_numbers, "*.chunks")
def split_problem (input_file_names, output_files):
    pass

```

This will split the incoming `input_file_names` into NNN number of *outputs* where NNN is not pre-determined:

The `output` (second) parameter of `@split` often contains a `glob` pattern like the `*.chunks` above.

Only **after** the task function has completed, will Ruffus match the **Output** parameter (`*.chunks`) against the files which have been created by `split_problem()` (e.g. `1.chunks`, `2.chunks`, `3.chunks`)

### 1.14.4 Be careful in specifying Output globs

Note that it is your responsibility to keep the **Output** specification tight enough so that Ruffus does not pick up extraneous files.

You can specify multiple *glob* patterns to match *all* the files which are the result of the splitting task function. These can even cover different directories, or groups of file names. This is a more extreme example:

```
@split("input.file", ['a*.bits', 'b*.pieces', 'somewhere_else/c*.stuff'])
def split_function (input_filename, output_files):
    "Code to split up 'input.file'"
```

### 1.14.5 Clean up previous pipeline runs

Problems arise when the current directory contains results of previous pipeline runs.

- For example, if the previous analysis involved a large data set, there might be 3 chunks: 1.chunks, 2.chunks, 3.chunks.
- In the current analysis, there might be a smaller data set which divides into only 2 chunks, 1.chunks and 2.chunks.
- Unfortunately, 3.chunks from the previous run is still hanging around and will be included erroneously by the glob \*.chunks.

**Warning: Your first duty in @split tasks functions should be to clean up**

To help you clean up thoroughly, Ruffus initialises the **output** parameter to all files which match specification.

The first order of business is thus invariably to cleanup ( delete with `os.unlink`) all files in **Output**.

```
#-----
#
#   split initial file
#
@split(create_random_numbers, "*.chunks")
def split_problem (input_file_names, output_files):
    """
        splits random numbers file into xxx files of chunk_size each
    """
    #
    #   clean up any files from previous runs
    #
    #for ff in glob.glob("*.chunks"):
    for ff in input_file_names:
        os.unlink(ff)
```

(The first time you run the example code, \*.chunks will initialise output\_files to an empty list. )

### 1.14.6 1 to many

`@split` is a one to many operator because its outputs are a list of *independent* items.

If `@split` generates 5 files, then this will lead to 5 jobs downstream.

This means we can just connect our old friend `@transform` to our pipeline and the results of `@split` will be analysed in parallel. This code should look familiar:

```
#-----
#      Calculate sum and sum of squares for each chunk file
#
@transform(split_problem, suffix(".chunks"), ".sums")
def sum_of_squares (input_file_name, output_file_name):
    pass
```

Which results in output like this:

```
>>> pipeline_run()
Job = [[random_numbers.list] -> *.chunks] completed
Completed Task = split_problem
    Job = [1.chunks -> 1.sum] completed
    Job = [10.chunks -> 10.sum] completed
    Job = [2.chunks -> 2.sum] completed
    Job = [3.chunks -> 3.sum] completed
    Job = [4.chunks -> 4.sum] completed
    Job = [5.chunks -> 5.sum] completed
    Job = [6.chunks -> 6.sum] completed
    Job = [7.chunks -> 7.sum] completed
    Job = [8.chunks -> 8.sum] completed
    Job = [9.chunks -> 9.sum] completed
Completed Task = sum_of_squares
```

Have a look at the *Example code for this chapter*

### 1.14.7 Nothing to many

Normally we would use `@originate` to create files from scratch, for example at the beginning of the pipeline.

However, sometimes, it is not possible to determine ahead of time how many files you will be creating from scratch. `@split` can also be useful even in such cases:

```
from random import randint
from ruffus import *
import os

# Create between 2 and 5 files
@split(None, "*start")
def create_initial_files(no_input_file, output_files):
    # cleanup first
    for oo in output_files:
        os.unlink(oo)
    # make new files
    for ii in range(randint(2,5)):
        open("%d.start" % ii, "w")

@transform(create_initial_files, suffix(".start"), ".processed")
def process_files(input_file, output_file):
    open(output_file, "w")

pipeline_run()
```

Giving:

```
>>> pipeline_run()
Job   = [None -> *.start] completed
Completed Task = create_initial_files
Job   = [0.start -> 0.processed] completed
Job   = [1.start -> 1.processed] completed
Completed Task = process_files
```

## 1.15 Chapter 13: @merge multiple input into a single result

See also:

- *Manual Table of Contents*
- *@merge* syntax
- *Example code for this chapter*

### 1.15.1 Overview of @merge

The *previous chapter* explained how **Ruffus** allows large jobs to be split into small pieces with *@split* and analysed in parallel using for example, our old friend *@transform*.

Having done this, our next task is to recombine the fragments into a seamless whole.

This is the role of the *@merge* decorator.

### 1.15.2 @merge is a many to one operator

*@transform* tasks multiple *inputs* and produces a single *output*, **Ruffus** is again agnostic as to the sort of data contained within this single *output*. It can be a single (string) file name, an arbitrary complicated nested structure with numbers, objects etc. Or even a list.

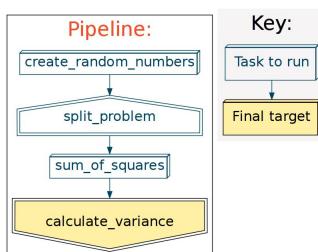
The main thing is that downstream tasks will interpret this output as a single entity leading to a single job.

*@split* and *@merge* are, in other words, about network topology.

Because of this *@merge* is also very useful for summarising the progress in our pipeline. At key selected points, we can gather data from the multitude of data or disparate *inputs* and *@merge* them to a single set of summaries.

### 1.15.3 Example: Combining partial solutions: Calculating variances

The *previous chapter* we had almost completed all the pieces of our flowchart:



What remains is to take the partial solutions from the different `.sums` files and turn these into the variance as follows:

```
variance = (sum_squared - sum * sum / N) / N
```

where `N` is the number of values

See the [wikipedia](#) entry for a discussion of why this is a very naive approach.

To do this, all we have to do is iterate through all the values in `*.sums`, add up the `sums` and `sum_squared`, and apply the above (naive) formula.

```
#  
#     @merge files together  
#  
@merge(sum_of_squares, "variance.result")  
def calculate_variance (input_file_names, output_file_name):  
    """  
    Calculate variance naively  
    """  
    #  
    #     initialise variables  
    #  
    all_sum_squared = 0.0  
    all_sum         = 0.0  
    all_cnt_values = 0.0  
    #  
    # added up all the sum_squared, and sum and cnt_values from all the chunks  
    #  
    for input_file_name in input_file_names:  
        sum_squared, sum, cnt_values = map(float, open(input_file_name).readlines())  
        all_sum_squared += sum_squared  
        all_sum         += sum  
        all_cnt_values += cnt_values  
    all_mean = all_sum / all_cnt_values  
    variance = (all_sum_squared - all_sum * all_mean) / (all_cnt_values)  
    #  
    #     print output  
    #  
    open(output_file_name, "w").write("%s\n" % variance)
```

This results in the following equivalent function call:

```
calculate_variance (["1.sums", "2.sums", "3.sums",  
                    "4.sums", "5.sums", "6.sums",  
                    "7.sums", "8.sums", "9.sums", "10.sums"], "variance.result")
```

and the following display:

```
>>> pipeline_run()  
Job = [[1.sums, 10.sums, 2.sums, 3.sums, 4.sums, 5.sums, 6.sums, 7.sums, 8.sums, 9.sums],  
      Completed Task = calculate_variance
```

The final result is in `variance.result`

Have a look at the *complete example code for this chapter*.

## 1.16 Chapter 14: Multiprocessing, drmaa and Computation Clusters

See also:

- *Manual Table of Contents*
  - *@jobs\_limit* syntax
  - *pipeline\_run()* syntax
  - *drmaa\_wrapper.run\_job()* syntax
- 

**Note:** Remember to look at the example code:

- *Chapter 14: Python Code for Multiprocessing, drmaa and Computation Clusters*
- 

### 1.16.1 Overview

#### Multi Processing

*Ruffus* uses python `multiprocessing` to run each job in a separate process.

This means that jobs do *not* necessarily complete in the order of the defined parameters. Task hierachies are, of course, inviolate: upstream tasks run before downstream, dependent tasks.

Tasks that are independent (i.e. do not precede each other) may be run in parallel as well.

The number of concurrent jobs can be set in *pipeline\_run*:

```
pipeline_run([parallel_task], multiprocess = 5)
```

If `multiprocess` is set to 1, then jobs will be run on a single process.

#### Data sharing

Running jobs in separate processes allows *Ruffus* to make full use of the multiple processors in modern computers. However, some `multiprocessing` guidelines should be borne in mind when writing *Ruffus* pipelines. In particular:

- Try not to pass large amounts of data between jobs, or at least be aware that this has to be marshalled across process boundaries.
- Only data which can be `pickled` can be passed as parameters to *Ruffus* task functions. Happily, that applies to almost any native Python data type. The use of the rare, unpicklable object will cause python to complain (fail) loudly when *Ruffus* pipelines are run.

### 1.16.2 Restricting parallelism with *@jobs\_limit*

Calling `pipeline_run(multiprocess = NNN)` allows multiple jobs (from multiple independent tasks) to be run in parallel. However, there are some operations that consume so many resources that we might want them to run with less or no concurrency.

For example, we might want to download some files via FTP but the server restricts requests from each IP address. Even if the rest of the pipeline is running 100 jobs in parallel, the FTP downloading must be restricted to 2 files at a time. We would really like to keep the pipeline running as is, but let this one operation run either serially, or with little concurrency.

- `pipeline_run(multiprocess = NNN)` sets the pipeline-wide concurrency but
- `@jobs_limit(MMM)` sets concurrency at MMM only for jobs in the decorated task.

The optional name (e.g. `@jobs_limit(3, "ftp_download_limit")`) allows the same limit to be shared across multiple tasks. To be pedantic: a limit of 3 jobs at a time would be applied across all tasks which have a `@jobs_limit` named "ftp\_download\_limit".

The *example code* uses up to 10 processes across the pipeline, but runs the `stage1_big` and `stage1_small` tasks 3 at a time (shared across both tasks). `stage2` jobs run 5 at a time.

### 1.16.3 Using drmaa to dispatch work to Computational Clusters or Grid engines from Ruffus jobs

Ruffus has been widely used to manage work on computational clusters or grid engines. Though Ruffus task functions cannot (yet!) run natively and transparently on remote cluster nodes, it is trivial to dispatch work across the cluster.

From version 2.4 onwards, Ruffus includes an optional helper module which interacts with [python bindings](#) for the widely used `drmaa` Open Grid Forum API specification. This allows jobs to dispatch work to a computational cluster and wait until it completes.

Here are the necessary steps

#### 1) Use a shared drmaa session:

Before your pipeline runs:

```
#  
#      start shared drmaa session for all jobs / tasks in pipeline  
#  
import drmaa  
drmaa_session = drmaa.Session()  
drmaa_session.initialize()
```

Cleanup after your pipeline completes:

```
#  
#      pipeline functions go here  
#  
if __name__ == '__main__':  
    drmaa_session.exit()
```

#### 2) import ruffus.drmaa\_wrapper

- The optional `ruffus.drmaa_wrapper` module needs to be imported explicitly:

```
# imported ruffus.drmaa_wrapper explicitly  
from ruffus.drmaa_wrapper import run_job, error_drmaa_job
```

#### 3) call drmaa\_wrapper.run\_job()

`drmaa_wrapper.run_job()` dispatches the work to a cluster node within a normal Ruffus job and waits for completion

This is the equivalent of `os.system` or `subprocess.check_output` but the code will run remotely as specified:

```
# ruffus.drmaa_wrapper.run_job
stdout_res, stderr_res = run_job(cmd_str
                                  job_name
                                  logger
                                  drmaa_session
                                  run_locally
                                  job_other_options
                                  = "touch " + output_file,
                                  = job_name,
                                  = logger,
                                  = drmaa_session,
                                  = options.local_run,
                                  = job_other_options)
```

The complete code is available [here](#)

- `drmaa_wrapper.run_job()` is a convenience wrapper around the python drmaa bindings `RunJob` function. It takes care of writing drmaa *job templates* for you.
- Each call creates a separate drmaa *job template*.

#### 4) Use multithread: `pipeline_run(multithread = NNN)`

**Warning:** `drmaa_wrapper.run_job()`  
requires `pipeline_run (multithread = NNN)`  
and will not work with `pipeline_run (multiprocess = NNN)`

##### Using multithreading rather than multiprocessing

- allows the drmaa session to be shared
- prevents “processing storms” which lock up the queue submission node when hundreds or thousands of grid engine / cluster commands complete at the same time.

```
pipeline_run (..., multithread = NNN, ...)
```

or if you are using ruffus.cmdline:

```
cmdline.run (options, multithread = options.jobs)
```

Normally multithreading reduces the amount of parallelism in python due to the python [Global interpreter Lock \(GIL\)](#). However, as the work load is almost entirely on another computer (i.e. a cluster / grid engine node) with a separate python interpreter, any cost benefit calculations of this sort are moot.

#### 5) Develop locally

`drmaa_wrapper.run_job()` provides two convenience parameters for developing grid engine pipelines:

- commands can run locally, i.e. on the local machine rather than on cluster nodes:

```
run_job(cmd_str, run_locally = True)
```

- Output files can be [touched](#), i.e. given the appearance of the work having been done without actually running the commands

```
run_job(cmd_str, touch_only = True)
```

## 1.16.4 Forcing a pipeline to appear up to date

Sometimes, we *know* that a pipeline has run to completion, that everything is up-to-date. However, Ruffus still insists on the basis of file modification times that you need to rerun.

For example, sometimes a trivial accounting modification needs to be made to a data file. Even though you know that this changes nothing in practice, Ruffus will detect the modification and ask to rerun everything from that point forwards.

One way to convince Ruffus that everything is fine is to manually `touch` all subsequent data files one by one in sequence so that the file timestamps follow the appropriate progression.

You can also ask *Ruffus* to do this automatically for you by running the pipeline in `touch` mode:

```
pipeline_run( touch_files_only = True)
```

`pipeline_run` will run your pipeline script normally working backwards from any specified final target, or else the last task in the pipeline. It works out where it should begin running, i.e. with the first out-of-date data files. After that point, instead of calling your pipeline task functions, each missing or out-of-date file is `touch-ed` in turn so that the file modification dates follow on successively.

This turns out to be useful way to check that your pipeline runs correctly by creating a series of dummy (empty files). However, *Ruffus* does not know how to read your mind to know which files to create from `@split` or `@subdivide` tasks.

Using `ruffus.cmdline` from version 2.4, you can just specify:

```
your script --touch_files_only [--other_options_of_your_own_etc]
```

## 1.17 Chapter 15: Logging progress through a pipeline

See also:

- *Manual Table of Contents*

---

**Note:** Remember to look at the *example code*

---

### 1.17.1 Overview

There are two parts to logging with **Ruffus**:

- Logging progress through the pipeline

This produces the sort of output displayed in this manual:

```
>>> pipeline_run([parallel_io_task])
Task = parallel_io_task
Job = ["a.1" -> "a.2", "A file"] completed
Job = ["b.1" -> "b.2", "B file"] unnecessary: already up to date
Completed Task = parallel_io_task
```

- Logging your own messages from within your pipelined functions.

Because **Ruffus** may run each task function in separate process on a separate CPU (multiprocessing), some attention has to be paid to how to send and synchronise your log messages across process boundaries.

We shall deal with these in turn.

### 1.17.2 Logging task/job completion

By default, *Ruffus* logs each task and each job as it is completed to `sys.stderr`.

By default, Ruffus logs to STDERR: `pipeline_run(logger = stderr_logger)`.

If you want to turn off all tracking messages as the pipeline runs, apart from setting `verbose = 0`, you can also use the aptly named Ruffus `black_hole_logger`:

```
pipeline_run(logger = black_hole_logger)
```

### Controlling logging verbosity

`pipeline_run()` currently has five levels of verbosity, set by the optional `verbose` parameter which defaults to 1:

```
verbose = 0: nothing  
verbose = 1: logs completed jobs/tasks;  
verbose = 2: logs up to date jobs in incomplete tasks  
verbose = 3: logs reason for running job  
verbose = 4: logs messages useful only for debugging ruffus pipeline code
```

`verbose > 5` are intended for debugging **Ruffus** by the developers and the details are liable to change from release to release

### 1.17.3 Use *ruffus.cmdline*

As always, it is easiest to use *ruffus.cmdline*.

Set your script to

- write messages to STDERR with the `--verbose` option and
- to a log file with the `--log_file` option.

```
from ruffus import *\n\n# Python logger which can be synchronised across concurrent Ruffus tasks\nlogger, logger_mutex = cmdline.setup_logging (__name__, options.log_file, options.verbose)\n\n@transform( ["job1.input"], suffix(".input"), ".output1")\ndef first_task(input_file, output_file):\n    pass\n\npipeline_run(logger=logger)
```

### 1.17.4 Customising logging

You can also specify exactly how logging works by providing a `logging` object to `pipeline_run()`. This log object should have `debug()` and `info()` methods.

Instead of writing your own, it is usually more convenient to use the python `logging` module which provides logging classes with rich functionality.

The *example code* sets up a logger to a rotating set of files

### 1.17.5 Log your own messages

You need to take a little care when logging your custom messages *within* your pipeline.

- If your Ruffus pipeline may run in parallel, make sure that logging is synchronised.
- If your Ruffus pipeline may run across separate processes, send your logging object across process boundaries.

logging objects can not be `pickled` and shared naively across processes. Instead, we need to create proxies which forward the logging to a single shared log.

The `ruffus.proxy_logger` module provides an easy way to share `logging` objects among jobs. This requires just two simple steps:

---

**Note:**

- This is a good template for sharing `non-picklable objects` across processes.
- 

#### 1. Set up logging

Things are easiest if you are using `ruffus.cmdline`:

```
# standard python logger which can be synchronised across concurrent Ruffus tasks
logger, logger_mutex = cmdline.setup_logging (__name__, options.log_file, options.verbose)
```

Otherwise, manually:

```
from ruffus.proxy_logger import *
(logger,
 logging_mutex) = make_shared_logger_and_proxy (setup_std_shared_logger,
                                                 "my_logger",
                                                 {"file_name": "/my/lg.log"})
```

#### 2. Share the proxy

Now, pass:

- `logger` (which forwards logging calls across jobs) and
- `logging_mutex` (which prevents different jobs which are logging simultaneously from being jumbled up)

to each job:

```
@transform( initial_file,
            suffix(".input"),
            ".output1",
            logger, logging_mutex),           # pass log and synchronisation as parameters
def first_task(input_file, output_file,
                logger, logging_mutex):        # pass log and synchronisation as parameters
    pass

# synchronise logging
```

```
with logging_mutex:  
    logger.info("Here we go logging...")
```

## 1.18 Chapter 16: @subdivide tasks to run efficiently and regroup with @collate

See also:

- *Manual Table of Contents*
- @subdivide syntax
- @collate syntax

### 1.18.1 Overview

In **Chapter 12** and **Chapter 13**, we saw how a large task can be @split into small jobs to be analysed efficiently in parallel. Ruffus can then @merge these back together to give a single, unified result.

This assumes that your pipeline is processing one item at a time. Usually, however, we will have, for example, 10 large pieces of data in play, each of which has to be subdivided into smaller pieces for analysis before being put back together.

This is the role of @subdivide and @subdivide.

Like @split, the number of output files @subdivide produces for *each Input* is not predetermined.

On the other hand, these output files should be named in such a way that they can later be grouped back together later using @subdivide.

This will be clearer with some worked examples.

### 1.18.2 @subdivide in parallel

Let us start from 3 files with varying number of lines. We wish to process these two lines at a time but we do not know ahead of time how long each file is:

```
from ruffus import *\nimport os, random, sys\n\n# Create files a random number of lines\n@originate(["a.start",\n            "b.start",\n            "c.start"])\n\ndef create_test_files(output_file):\n    cnt_lines = random.randint(1,3) * 2\n    with open(output_file, "w") as oo:\n        for ii in range(cnt_lines):\n            oo.write("data item = %d\n" % ii)\n    print "%s has %d lines" % (output_file, cnt_lines)\n\n#\n#    subdivide the input files into NNN fragment files of 2 lines each\n#
```

```

@subdivide( create_test_files,
            formatter(),
            "{path[0]}/{basename[0]}.*.fragment",
            "{path[0]}/{basename[0]}")
def subdivide_files(input_file, output_files, output_file_name_stem):
    #
    #   cleanup any previous results
    #
    for oo in output_files:
        os.unlink(oo)
    #
    #   Output files contain two lines each
    #       (new output files every even line)
    #
    cnt_output_files = 0
    for ii, line in enumerate(open(input_file)):
        if ii % 2 == 0:
            cnt_output_files += 1
            output_file_name = "%s.%d.fragment" % (output_file_name_stem, cnt_output_files)
            output_file = open(output_file_name, "w")
            print "          Subdivide %s -> %s" % (input_file, output_file_name)
            output_file.write(line)

    #
    #   Analyse each fragment independently
    #
@transform(subdivide_files, suffix(".fragment"), ".analysed")
def analyse_fragments(input_file, output_file):
    print "          Analysing %s -> %s" % (input_file, output_file)
    with open(output_file, "w") as oo:
        for line in open(input_file):
            oo.write("analysed " + line)

```

This produces the following output:

```

>>> pipeline_run(verbose = 1)
    a.start has 2 lines
    Job  = [None -> a.start] completed
    b.start has 6 lines
    Job  = [None -> b.start] completed
    c.start has 6 lines
    Job  = [None -> c.start] completed
Completed Task = create_test_files

    Subdivide a.start -> /home/lg/temp/a.1.fragment
    Job  = [a.start -> a.*.fragment, a] completed

    Subdivide b.start -> /home/lg/temp/b.1.fragment
    Subdivide b.start -> /home/lg/temp/b.2.fragment
    Subdivide b.start -> /home/lg/temp/b.3.fragment
    Job  = [b.start -> b.*.fragment, b] completed

    Subdivide c.start -> /home/lg/temp/c.1.fragment
    Subdivide c.start -> /home/lg/temp/c.2.fragment
    Subdivide c.start -> /home/lg/temp/c.3.fragment
    Job  = [c.start -> c.*.fragment, c] completed

```

```

Completed Task = subdivide_files

    Analysing /home/lg/temp/a.1.fragment -> /home/lg/temp/a.1.analysed
    Job   = [a.1.fragment -> a.1.analysed] completed
    Analysing /home/lg/temp/b.1.fragment -> /home/lg/temp/b.1.analysed
    Job   = [b.1.fragment -> b.1.analysed] completed

[ ...SEE EXAMPLE CODE FOR MORE LINES ...]

```

Completed Task = analyse\_fragments

a.start has two lines and results in a single .fragment file, while there are 3 b.\*.fragment files because it has 6 lines. Whatever their origin, all of the different fragment files are treated equally in analyse\_fragments() and processed (in parallel) in the same way.

### 1.18.3 Grouping using @collate

All that is left in our example is to reassemble the analysed fragments back together into 3 sets of results corresponding to the original 3 pieces of starting data.

This is straightforward by eye: the file names all have the same pattern: [abc] .\*.analysed:

```

a.1.analysed      ->  a.final_result
b.1.analysed      ->  b.final_result
b.2.analysed      ->  ..
b.3.analysed      ->  ..
c.1.analysed      ->  c.final_result
c.2.analysed      ->  ..

```

@collate does something similar:

1. Specify a string substitution e.g. c.???.analysed -> c.final\_result and
2. Ask ruffus to group together any **Input** (e.g. c.1.analysed, c.2.analysed) that will result in the same **Output** (e.g. c.final\_result)

```

#
#   ''XXX.???.analysed -> XXX.final_result''
#   Group results using original names
#
@collate(  analyse_fragments,
            # split file name into [abc].NUMBER.analysed
            formatter("/(?:P<NAME>[abc]+)\.\d+\.analysed$"),
            "{path[0]}/{NAME[0]}.final_result")
def recombine_analyses(input_file_names, output_file):
    with open(output_file, "w") as oo:
        for input_file in input_file_names:
            print "      Recombine %s -> %s" % (input_file, output_file)
            for line in open(input_file):
                oo.write(line)

```

This produces the following output:

```

Recombine /home/lg/temp/a.1.analysed -> /home/lg/temp/a.final_result
Job   = [[a.1.analysed] -> a.final_result] completed
Recombine /home/lg/temp/b.1.analysed -> /home/lg/temp/b.final_result
Recombine /home/lg/temp/b.2.analysed -> /home/lg/temp/b.final_result

```

```

Recombine /home/lg/temp/b.3.analysed -> /home/lg/temp/b.final_result
Job  = [[b.1.analysed, b.2.analysed, b.3.analysed] -> b.final_result] completed
Recombine /home/lg/temp/c.1.analysed -> /home/lg/temp/c.final_result
Recombine /home/lg/temp/c.2.analysed -> /home/lg/temp/c.final_result
Recombine /home/lg/temp/c.3.analysed -> /home/lg/temp/c.final_result
Job  = [[c.1.analysed, c.2.analysed, c.3.analysed] -> c.final_result] completed
Completed Task = recombine_analyses

```

**Warning:**

- **Input** file names are grouped together not in a guaranteed order.  
For example, the fragment files may not be sent to `recombine_analyses(input_file_names, ...)` in alphabetically or any other useful order.  
You may want to sort **Input** before concatenation.
- All **Input** are grouped together if they have both the same **Output** and **Extra** parameters. If any string substitution is specified in any of the other **Extra** parameters to `@subdivide`, they must give the same answers for **Input** in the same group.

## 1.19 Chapter 17: `@combinations`, `@permutations` and all versus all `@product`

**See also:**

- *Manual Table of Contents*
- `@combinations_with_replacement`
- `@combinations`
- `@permutations`
- `@product`
- `formatter()`

**Note:** Remember to look at the example code:

- *Chapter 17: Python Code for @combinations, @permutations and all versus all @product*

### 1.19.1 Overview

A surprising number of computational problems involve some sort of all versus all calculations. Previously, this would have required all the parameters to be supplied using a custom function on the fly with `@files`.

From version 2.4, *Ruffus* supports `@combinations_with_replacement`, `@combinations`, `@permutations`, `@product`.

These provide as far as possible all the functionality of the four combinatorics iterators from the standard python `itertools` functions of the same name.

## 1.19.2 Generating output with *formatter()*

String replacement always takes place via *formatter()*. Unfortunately, the other *Ruffus* workhorses of *regex()* and *suffix()* do not have sufficient syntactic flexibility.

Each combinatorics decorator deals with multiple sets of inputs whether this might be:

- a self-self comparison (such as `@combinations_with_replacement`, `@combinations`, `@permutations`) or,
- a self-other comparison (`@product`)

The replacement strings thus require an extra level of indirection to refer to parsed components.

1. The first level refers to which *set* of inputs.
2. The second level refers to which input file in any particular *set* of inputs.

For example, if the *inputs* are `[A1,A2],[B1,B2],[C1,C2]` vs `[P1,P2],[Q1,Q2],[R1,R2]` vs `[X1,X2],[Y1,Y2],[Z1,Z2]`, then '`{basename[2][0]}`' is the *basename* for

- the third set of inputs (**X,Y,Z**) and
- the first file name string in each **Input** of that set (**X1, Y1, Z1**)

## 1.19.3 All vs all comparisons with *@product*

*@product* generates the Cartesian **product** between sets of input files, i.e. all vs all comparisons.

The effect is analogous to a nested for loop.

*@product* can be useful, for example, in bioinformatics for finding the corresponding genes (orthologues) for a set of proteins in multiple species.

```
>>> from itertools import product
>>> # product('ABC', 'XYZ') --> AX AY AZ BX BY BZ CX CY CZ
>>> [ "".join(a) for a in product('ABC', 'XYZ') ]
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

This example Calculates the *@product* of **A,B** and **P,Q** and **X,Y** files

```
from ruffus import *
from ruffus.combinatorics import *

# Three sets of initial files
@originate([ 'a.start', 'b.start' ])
def create_initial_files_ab(output_file):
    with open(output_file, "w") as oo: pass

@originate([ 'p.start', 'q.start' ])
def create_initial_files_pq(output_file):
    with open(output_file, "w") as oo: pass

@originate([ ['x.1_start', 'x.2_start'],
             ['y.1_start', 'y.2_start'] ])
def create_initial_files_xy(output_file):
    with open(output_file, "w") as oo: pass

# @product
@product(   create_initial_files_ab,           # Input
          formatter("(start)$"),            # match input file set # 1
```

```

create_initial_files_pq,           # Input
formatter("(start)$"),           # match input file set # 2

create_initial_files_xy,          # Input
formatter("(start)$"),           # match input file set # 3

"{path[0][0]}/"                  # Output Replacement string
"{basename[0][0]}_vs_"
"{basename[1][0]}_vs_"
"{basename[2][0]}.product",       #

"{path[0][0]}",                  # Extra parameter: path for 1st set of files, 1st

["{basename[0][0]}",             # Extra parameter: basename for 1st set of files
 "{basename[1][0]}",             # 2nd
 "{basename[2][0]}",             # 3rd
])

def product_task(input_file, output_parameter, shared_path, basenames):
    print "# basenames = ", ".join(basenames)
    print "input_parameter = ", input_file
    print "output_parameter = ", output_parameter, "\n"

#
#      Run
#
pipeline_run(verbose=0)

```

This results in:

```

>>> pipeline_run(verbose=0)

# basenames = a p x
input_parameter = ('a.start', 'p.start', 'x.start')
output_parameter = /home/lg/temp/a_vs_p_vs_x.product

# basenames = a p y
input_parameter = ('a.start', 'p.start', 'y.start')
output_parameter = /home/lg/temp/a_vs_p_vs_y.product

# basenames = a q x
input_parameter = ('a.start', 'q.start', 'x.start')
output_parameter = /home/lg/temp/a_vs_q_vs_x.product

# basenames = a q y
input_parameter = ('a.start', 'q.start', 'y.start')
output_parameter = /home/lg/temp/a_vs_q_vs_y.product

# basenames = b p x
input_parameter = ('b.start', 'p.start', 'x.start')
output_parameter = /home/lg/temp/b_vs_p_vs_x.product

# basenames = b p y
input_parameter = ('b.start', 'p.start', 'y.start')
output_parameter = /home/lg/temp/b_vs_p_vs_y.product

# basenames = b q x
input_parameter = ('b.start', 'q.start', 'x.start')

```

```
output_parameter = /home/lg/temp/b_vs_q_vs_x.product
# basenames      = b q y
input_parameter = ('b.start', 'q.start', 'y.start')
output_parameter = /home/lg/temp/b_vs_q_vs_y.product
```

#### 1.19.4 Permute all k-tuple orderings of inputs without repeats using `@permutations`

Generates the permutations for all the elements of a set of Input (e.g. A B C D),

- r-length tuples of *input* elements
- excluding repeated elements (A A)
- and order of the tuples is significant (both A B and B A).

```
>>> from itertools import permutations
>>> # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
>>> [ "".join(a) for a in permutations("ABCD", 2)]
['AB', 'AC', 'AD', 'BA', 'BC', 'BD', 'CA', 'CB', 'CD', 'DA', 'DB', 'DC']
```

This following example calculates the `@permutations` of A,B,C,D files

```
from ruffus import *
from ruffus.combinatorics import *

# initial file pairs
@originate([
    ['A.1_start', 'A.2_start'],
    ['B.1_start', 'B.2_start'],
    ['C.1_start', 'C.2_start'],
    ['D.1_start', 'D.2_start']])
def create_initial_files_ABCD(output_files):
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

    # @permutations
    @permutations(create_initial_files_ABCD,           # Input
                  formatter(),                      # match input files

                  # tuple of 2 at a time
                  2,

                  # Output Replacement string
                  "{path[0][0]}/"
                  "{basename[0][1]}_vs_"
                  "{basename[1][1]}.permutations",

                  # Extra parameter: path for 1st set of files, 1st file name
                  "{path[0][0]}",

                  # Extra parameter
                  [{"basename[0][0]": "# basename for 1st set of files, 1st file name",
                    "basename[1][0]": "#                                2nd
                  }])

    def permutations_task(input_file, output_parameter, shared_path, basenames):
        print " - ".join(basenames)
```

---

```

#
#           Run
#
pipeline_run(verbose=0)

```

This results in:

```

>>> pipeline_run(verbose=0)

A - B
A - C
A - D
B - A
B - C
B - D
C - A
C - B
C - D
D - A
D - B
D - C

```

### 1.19.5 Select unordered k-tuples within inputs excluding repeated elements using @combinations

Generates the combinations for all the elements of a set of Input (e.g. A B C D),

- r-length tuples of *input* elements
- without repeated elements (A A)
- where order of the tuples is irrelevant (either A B or B A, not both).

@combinations can be useful, for example, in calculating a transition probability matrix for a set of states. The diagonals are meaningless “self-self” transitions which are excluded.

```

>>> from itertools import combinations
>>> # combinations('ABCD', 3) --> ABC ABD ACD BCD
>>> [ "".join(a) for a in combinations("ABCD", 3) ]
['ABC', 'ABD', 'ACD', 'BCD']

```

This example calculates the @combinations of A,B,C,D files

```

from ruffus import *
from ruffus.combinatorics import *

# initial file pairs
@originate([
    ['A.1_start', 'A.2_start'],
    ['B.1_start', 'B.2_start'],
    ['C.1_start', 'C.2_start'],
    ['D.1_start', 'D.2_start']])
def create_initial_files_ABCD(output_files):
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

# @combinations
@combinations(create_initial_files_ABCD,          # Input
              formatter(),                      # match input files

```

```
# tuple of 3 at a time
3,

# Output Replacement string
"{}[0][0]}/"
"{}[0][1]}_vs_"
"{}[1][1]}_vs_"
"{}[2][1]}.combinations",

# Extra parameter: path for 1st set of files, 1st file name
"{}[0][0]",

# Extra parameter
["{}[0][0]", # basename for 1st set of files, 1st file name
 "{}[1][0]", # 2nd
 "{}[2][0]", # 3rd
 ])

def combinations_task(input_file, output_parameter, shared_path, basenames):
    print " - ".join(basenames)

#
#       Run
#
pipeline_run(verbose=0)
```

This results in:

```
>>> pipeline_run(verbose=0)
A - B - C
A - B - D
A - C - D
B - C - D
```

### 1.19.6 Select unordered k-tuples within inputs *including* repeated elements with @combinations\_with\_replacement

Generates the combinations\_with\_replacement for all the elements of a set of Input (e.g. A B C D),

- r-length tuples of *input* elements
- including repeated elements (A A)
- where order of the tuples is irrelevant (either A B or B A, not both).

@combinations\_with\_replacement can be useful, for example, in bioinformatics for finding evolutionary relationships between genetic elements such as proteins and genes. Self-self comparisons can be used a baseline for scaling similarity scores.

```
>>> from itertools import combinations_with_replacement
>>> # combinations_with_replacement('ABCD', 2) --> AA AB AC AD BB BC BD CC CD DD
>>> [ "".join(a) for a in combinations_with_replacement('ABCD', 2) ]
['AA', 'AB', 'AC', 'AD', 'BB', 'BC', 'BD', 'CC', 'CD', 'DD']
```

This example calculates the @combinations\_with\_replacement of A,B,C,D files

```

from ruffus import *
from ruffus.combinatorics import *

# initial file pairs
@originate([ ['A.1_start', 'A.2_start'],
             ['B.1_start', 'B.2_start'],
             ['C.1_start', 'C.2_start'],
             ['D.1_start', 'D.2_start']])
def create_initial_files_ABCD(output_files):
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

# @combinations_with_replacement
@combinations_with_replacement(create_initial_files_ABCD,           # Input
                                 formatter(),                      # match input files
                                 tuple_size=2,                     # tuple of 2 at a time
                                 tuple_index=2,
                                 output_replacement_string=
                                 "{path[0][0]}/"
                                 "{basename[0][1]}_vs_"
                                 "{basename[1][1]}.combinations_with_replacement",
                                 extra_parameter=path[0][0],
                                 extra_parameter_desc="Extra parameter: path for 1st set of files, 1st file name",
                                 extra_parameter_value=path[0][0],
                                 extra_parameter_desc="Extra parameter",
                                 extra_parameter_value=[basename[0][0], basename[1][0]],
                                 extra_parameter_desc="basename for 1st set of files, 1st file name",
                                 extra_parameter_value=basename[0][0],
                                 extra_parameter_desc="2nd",
                                 extra_parameter_value=basename[1][0])
def combinations_with_replacement_task(input_file, output_parameter, shared_path, basenames):
    print " - ".join(basenames)

#
#       Run
#
pipeline_run(verbose=0)

```

This results in:

```

>>> pipeline_run(verbose=0)
A - A
A - B
A - C
A - D
B - B
B - C
B - D
C - C
C - D
D - D

```

## 1.20 Chapter 18: Turning parts of the pipeline on and off at runtime with `@active_if`

See also:

- *Manual Table of Contents*
- *@active\_if syntax in detail*

### 1.20.1 Overview

It is sometimes useful to be able to switch on and off parts of a pipeline. For example, a pipeline might have two difference code paths depending on the type of data it is being asked to analyse.

One surprisingly easy way to do this is to use a python `if` statement around particular task functions:

```
from ruffus import *

run_task1 = True

@originate(['a.foo', 'b.foo'])
def create_files(output_file):
    open(output_file, "w")

if run_task1:
    # might not run
    @transform(create_files, suffix(".foo"), ".bar")
    def foobar(input_file, output_file):
        open(output_file, "w")

    @transform(foobar, suffix(".bar"), ".result")
    def wrap_up(input_file, output_file):
        open(output_file, "w")

pipeline_run()
```

This simple solution has a number of drawbacks:

1. The on/off decision is a one off event that happens when the script is loaded. Ideally, we would like some flexibility, and postpone the decision until `pipeline_run()` is invoked.
2. When `if` is false, the entire task function becomes invisible, and if there are any downstream tasks, as in the above example, *Ruffus* will complain loudly about missing dependencies.

### 1.20.2 `@active_if` controls the state of tasks

- Switches tasks on and off at run time depending on its parameters
- Evaluated each time `pipeline_run`, `pipeline_printout` or `pipeline_printout_graph` is called.
- Dormant tasks behave as if they are up to date and have no output.

The Design and initial implementation were contributed by Jacob Biesinger

The following example shows its flexibility and syntax:

```

from ruffus import *
run_if_true_1 = True
run_if_true_2 = False
run_if_true_3 = True

#
#      task1
#
@originate(['a.foo', 'b.foo'])
def create_files(outfile):
    """
    create_files
    """
    open(outfile, "w").write(outfile + "\n")

#
# Only runs if all three run_if_true conditions are met
#
# @active_if determines if task is active
@active_if(run_if_true_1, lambda: run_if_true_2)
@active_if(run_if_true_3)
@transform(create_files, suffix(".foo"), ".bar")
def this_task_might_be_inactive(infile, outfile):
    open(outfile, "w").write("%s -> %s\n" % (infile, outfile))

# @active_if switches off task because run_if_true_2 == False
pipeline_run(verbose = 3)

# @active_if switches on task because all run_if_true conditions are met
run_if_true_2 = True
pipeline_run(verbose = 3)

```

The task starts off inactive:

```

>>> # @active_if switches off task "this_task_might_be_inactive" because run_if_true_2 == False
>>> pipeline_run(verbose = 3)

Task enters queue = create_files
create_files
    Job = [None -> a.foo] Missing file [a.foo]
    Job = [None -> b.foo] Missing file [b.foo]
    Job = [None -> a.foo] completed
    Job = [None -> b.foo] completed
Completed Task = create_files
Inactive Task = this_task_might_be_inactive

```

Now turn on the task:

```

>>> # @active_if switches on task "this_task_might_be_inactive" because all run_if_true conditions are met
>>> run_if_true_2 = True
>>> pipeline_run(verbose = 3)

Task enters queue = this_task_might_be_inactive

    Job = [a.foo -> a.bar] Missing file [a.bar]
    Job = [b.foo -> b.bar] Missing file [b.bar]

```

```
Job   = [a.foo -> a.bar] completed
Job   = [b.foo -> b.bar] completed
Completed Task = this_task_might_be_inactive
```

## 1.21 Chapter 19: Signal the completion of each stage of our pipeline with `@posttask`

See also:

- *Manual Table of Contents*
- `@posttask` syntax

### 1.21.1 Overview

It is often useful to signal the completion of each task by specifying a specific action to be taken or function to be called. This can range from printing out some message, or [touching](#) some sentinel file, to emailing the author. This is particular useful if the *task* is a recipe apply to an unspecified number of parameters in parallel in different *jobs*. If the task is never run, or if it fails, needless-to-say no task completion action will happen.

*Ruffus* uses the `@posttask` decorator for this purpose.

#### `@posttask`

We can signal the completion of each task by specifying one or more function(s) using `@posttask`

```
from ruffus import *

def task_finished():
    print "hooray"

@posttask(task_finished)
@originate("a.1")
def create_if_necessary(output_file):
    open(output_file, "w")

pipeline_run([create_if_necessary])
```

This is such a short function, we might as well write it in-line:

```
@posttask(lambda: sys.stdout.write("hooray\n"))
@originate("a.1")
def create_if_necessary(output_file):
    open(output_file, "w")
```

---

**Note:** The function(s) provided to `@posttask` will be called if the pipeline passes through a task, even if none of its jobs are run because they are up-to-date. This happens when an upstream task is out-of-date, and the execution passes through this point in the pipeline. See the example in *Appendix 2: How dependency is checked* of this manual.

---

## ***touch\_file***

One way to note the completion of a task is to create some sort of “flag” file. Each stage in a traditional make pipeline would contain a `touch completed.flag`.

This is such a useful idiom that *Ruffus* provides the shorthand `touch_file`:

```
from ruffus import *

@posttask(touch_file("task_completed.flag"))
@files(None, "a.1")
def create_if_necessary(input_file, output_file):
    open(output_file, "w")

pipeline_run()
```

## **Adding several post task actions**

You can, of course, add more than one different action to be taken on completion of the task, either by stacking up as many `@posttask` decorators as necessary, or by including several functions in the same `@posttask`:

```
from ruffus import *

@posttask(print_hooray, print_whoppee)
@posttask(print_hip_hip, touch_file("sentinel_flag"))
@originate("a.1")
def create_if_necessary(output_file):
    open(output_file, "w")

pipeline_run()
```

## **1.22 Chapter 20: Manipulating task inputs via string substitution using `inputs()` and `add_inputs()`**

See also:

- *Manual Table of Contents*
- `inputs()` syntax
- `add_inputs()` syntax

---

**Note:** Remember to look at the example code:

- *Chapter 20: Python Code for Manipulating task inputs via string substitution using `inputs()` and `add_inputs()`*
- 

### **1.22.1 Overview**

The previous chapters have been described how *Ruffus* allows the **Output** names for each job to be generated from the *Input* names via string substitution. This is how *Ruffus* can automatically chain multiple tasks in a pipeline together seamlessly.

Sometimes it is useful to be able to modify the **Input** by string substitution as well. There are two situations where this additional flexibility is needed:

1. You need to add additional prerequisites or filenames to the **Input** of every single job
2. You need to add additional **Input** file names which are some variant of the existing ones.

Both will be much more obvious with some examples

## 1.22.2 Adding additional *input* prerequisites per job with *add\_inputs()*

### 1. Example: compiling c++ code

Let us first compile some c++ ("\*.cpp") files using plain @transform syntax:

```
# source files exist before our pipeline
source_files = ["hasty.cpp", "tasty.cpp", "messy.cpp"]
for source_file in source_files:
    open(source_file, "w")

from ruffus import *

@transform(source_files, suffix(".cpp"), ".o")
def compile(input_filename, output_file):
    open(output_file, "w")

pipeline_run()
```

### 2. Example: Adding a common header file with *add\_inputs()*

```
# source files exist before our pipeline
source_files = ["hasty.cpp", "tasty.cpp", "messy.cpp"]
for source_file in source_files:
    open(source_file, "w")

# common (universal) header exists before our pipeline
open("universal.h", "w")

from ruffus import *

# make header files
@transform(source_files, suffix(".cpp"), ".h")
def create_matching_headers(input_file, output_file):
    open(output_file, "w")

@transform(source_files, suffix(".cpp"),
          # add header to the input of every job
          add_inputs("universal.h",
                     # add result of task create_matching_headers to the input of every job
                     create_matching_headers),
          ".o")
def compile(input_filename, output_file):
    open(output_file, "w")

pipeline_run()

>>> pipeline_run()
```

```

Job   = [hasty.cpp -> hasty.h] completed
Job   = [messy.cpp -> messy.h] completed
Job   = [tasty.cpp -> tasty.h] completed
Completed Task = create_matching_headers
    Job = [[hasty.cpp, universal.h, hasty.h, messy.h, tasty.h] -> hasty.o] completed
    Job = [[messy.cpp, universal.h, hasty.h, messy.h, tasty.h] -> messy.o] completed
    Job = [[tasty.cpp, universal.h, hasty.h, messy.h, tasty.h] -> tasty.o] completed
Completed Task = compile

```

### 3. Example: Additional *Input* can be tasks

We can also add a task name to `add_inputs()`. This chains the **Output**, i.e. run time results, of any previous task as an additional **Input** to every single job in the task.

```

# make header files
@transform(source_files, suffix(".cpp"), ".h")
def create_matching_headers(input_file, output_file):
    open(output_file, "w")

@transform(source_files,
          # add header to the input of every job
          add_inputs("universal.h",
                     # add result of task create_matching_headers to the input of every job
                     create_matching_headers),
          ".o")
def compile(input_filenames, output_file):
    open(output_file, "w")

pipeline_run()

>>> pipeline_run()
Job   = [[hasty.cpp, universal.h, hasty.h, messy.h, tasty.h] -> hasty.o] completed
Job   = [[messy.cpp, universal.h, hasty.h, messy.h, tasty.h] -> messy.o] completed
Job   = [[tasty.cpp, universal.h, hasty.h, messy.h, tasty.h] -> tasty.o] completed
Completed Task = compile

```

### 4. Example: Add corresponding files using `add_inputs()` with *formatter* or *regex*

The previous example created headers corresponding to our source files and added them as the **Input** to the compilation. That is generally not what you want. Instead, what is generally need is a way to

1. Look up the exact corresponding header for the *specific* job, and not add all possible files to all jobs in a task. When compiling `hasty.cpp`, we just need to add `hasty.h` (and `universal.h`).
2. Add a pre-existing file name (`hasty.h` already exists. Don't create it via another task.)

This is a surprisingly common requirement: In bioinformatics sometimes DNA or RNA sequence files come singly in `*.fastq` and sometimes in `matching pairs`: `*1.fastq`, `*2.fastq` etc. In the latter case, we often need to make sure that both sequence files are being processed in tandem. One way is to take one file name (`*1.fastq`) and look up the other.

`add_inputs()` uses standard *Ruffus* string substitution via *formatter* and *regex* to lookup (generally) **Input** file names. (As a rule *suffix* only substitutes **Output** file names.)

```

@transform( source_files,
            formatter(".cpp$"),
            # corresponding header for each source file

```

```

        add_inputs("{basename[0]}.h",
                   # add header to the input of every job
                   "universal.h"),
        "{basename[0]}.o")
def compile(input_filenames, output_file):
    open(output_file, "w")

```

This script gives the following output

```

>>> pipeline_run()
Job   = [[hasty.cpp, hasty.h, universal.h] -> hasty.o] completed
Job   = [[messy.cpp, messy.h, universal.h] -> messy.o] completed
Job   = [[tasty.cpp, tasty.h, universal.h] -> tasty.o] completed
Completed Task = compile

```

### 1.22.3 Replacing all input parameters with *inputs()*

The previous examples all *added* to the set of **Input** file names. Sometimes it is necessary to replace all the **Input** parameters altogether.

#### 5. Example: Running matching python scripts using *inputs()*

Here is a contrived example: we wish to find all cython/python files which have been compiled into corresponding c++ source files. Instead of compiling the c++, we shall invoke the corresponding python scripts.

Given three c++ files and their corresponding python scripts:

```

@transform( source_files,
            formatter(".cpp$"),
            # corresponding python file for each source file
            inputs("{basename[0]}.py"),
            "{basename[0]}.results")
def run_corresponding_python(input_filenames, output_file):
    open(output_file, "w")

```

The *Ruffus* code will call each python script corresponding to their c++ counterpart:

```

>>> pipeline_run()
Job   = [hasty.py -> hasty.results] completed
Job   = [messy.py -> messy.results] completed
Job   = [tasty.py -> tasty.results] completed
Completed Task = run_corresponding_python

```

### 1.23 Chapter 21: Esoteric: Generating parameters on the fly with *@files*

See also:

- *Manual Table of Contents*
- *@files on-the-fly syntax in detail*

---

**Note:** Remember to look at the example code:

- Chapter 21: Esoteric: Python Code for Generating parameters on the fly with @files
- 

## 1.23.1 Overview

The different *Ruffus decorators* connect up different tasks and generate *Output* (file names) from your *Input* in all sorts of different ways.

However, sometimes, none of them *quite* do exactly what you need. And it becomes necessary to generate your own *Input* and *Output* parameters on the fly.

Although this additional flexibility comes at the cost of a lot of extra inconvenient code, you can continue to leverage the rest of *Ruffus* functionality such as checking whether files are up to date or not.

## 1.23.2 @files syntax

To generate parameters on the fly, use the `@files` with a *generator function* which yields one list / tuple of parameters per job.

For example:

```
from ruffus import *

# generator function
def generate_parameters_on_the_fly():
    """
    returns one list of parameters per job
    """
    parameters = [
        ['A.input', 'A.output', (1, 2)], # 1st job
        ['B.input', 'B.output', (3, 4)], # 2nd job
        ['C.input', 'C.output', (5, 6)], # 3rd job
    ]
    for job_parameters in parameters:
        yield job_parameters

# tell ruffus that parameters should be generated on the fly
@files(generate_parameters_on_the_fly)
def pipeline_task(input, output, extra):
    open(output, "w").write(open(input).read())
    sys.stderr.write("%d + %d => %d\n" % (extra[0], extra[1], extra[0] + extra[1]))

pipeline_run()
```

Produces:

**Task = parallel\_task** 1 + 2 = 3 Job = ["A", 1, 2] completed 3 + 4 = 7 Job = ["B", 3, 4]  
completed 5 + 6 = 11 Job = ["C", 5, 6] completed

---

**Note:** Be aware that the parameter generating function may be invoked *more than once*: \* The first time to check if this part of the pipeline is up-to-date. \* The second time when the pipeline task function is run.

---

The resulting custom *inputs*, *outputs* parameters per job are treated normally for the purposes of checking to see if jobs are up-to-date and need to be re-run.

### 1.23.3 A Cartesian Product, all vs all example

The *accompanying example* provides a more realistic reason why you would want to generate parameters on the fly. It is a fun piece of code, which generates  $N \times M$  combinations from two sets of files as the *inputs* of a pipeline stage.

The *inputs / outputs* filenames are generated as a pair of nested for-loops to produce the  $N$  (outside loop)  $\times M$  (inside loop) combinations, with the appropriate parameters for each job yielded per iteration of the inner loop. The gist of this is:

```
#_
#   Generator function
#
#       N x M jobs
#_
def generate_simulation_params ():

    """
    Custom function to generate
    file names for gene/gwas simulation study
    """

    for sim_file in get_simulation_files():
        for (gene, gwas) in get_gene_gwas_file_pairs():
            result_file = "%s.%s.results" % (gene, sim_file)
            yield (gene, gwas, sim_file), result_file

@files(generate_simulation_params)
def gwas_simulation(input_files, output_file):
    ...

If get_gene_gwas_file_pairs() produces:
```

```
['a.sim', 'b.sim', 'c.sim']
```

and get\_gene\_gwas\_file\_pairs() produces:

```
[('1.gene', '1.gwas'), ('2.gene', '2.gwas')]
```

then we would end up with  $3 \times 2 = 6$  jobs and the following equivalent function calls:

```
gwas_simulation(('1.gene', '1.gwas', 'a.sim'), "1.gene.a.sim.results")
gwas_simulation(('2.gene', '2.gwas', 'a.sim'), "2.gene.a.sim.results")
gwas_simulation(('1.gene', '1.gwas', 'b.sim'), "1.gene.b.sim.results")
gwas_simulation(('2.gene', '2.gwas', 'b.sim'), "2.gene.b.sim.results")
gwas_simulation(('1.gene', '1.gwas', 'c.sim'), "1.gene.c.sim.results")
gwas_simulation(('2.gene', '2.gwas', 'c.sim'), "2.gene.c.sim.results")
```

The *accompanying code* looks slightly more complicated because of some extra bookkeeping.

You can compare this approach with the alternative of using `@product`:

---

```

#_
#
#      N x M jobs
#_
@product(    os.path.join(simulation_data_dir, "*simulation"),
              formatter(),

              os.path.join(gene_data_dir, "*.gene"),
              formatter(),

              # add gwas as an input: looks like *.gene but with a differnt extension
              add_inputs("{path[1][0]}/{basename[1][0]}.gwas")

              "{basename[0][0]}.{basename[1][0]}.results")      # output file
def gwas_simulation(input_files, output_file):
    ...

```

## 1.24 Chapter 22: Esoteric: Running jobs in parallel without files using @parallel

See also:

- *Manual Table of Contents*
- @parallel syntax in detail

### 1.24.1 @parallel

@parallel supplies parameters for multiple **jobs** exactly like @files except that:

1. The first two parameters are not treated like *inputs* and *outputs* parameters, and strings are not assumed to be file names
2. Thus no checking of whether each job is up-to-date is made using *inputs* and *outputs* files
3. No expansions of *glob* patterns or *output* from previous tasks is carried out.

This syntax is most useful when a pipeline stage does not involve creating or consuming any files, and you wish to forego the conveniences of @files, @transform etc.

The following code performs some arithmetic in parallel:

```

import sys
from ruffus import *
parameters = [
    ['A', 1, 2], # 1st job
    ['B', 3, 4], # 2nd job
    ['C', 5, 6], # 3rd job
]
@parallel(parameters)
def parallel_task(name, param1, param2):
    sys.stderr.write("    Parallel task %s: " % name)
    sys.stderr.write("%d + %d = %d\n" % (param1, param2, param1 + param2))

pipeline_run([parallel_task])

```

produces the following:

```
Task = parallel_task
Parallel task A: 1 + 2 = 3
Job = ["A", 1, 2] completed
Parallel task B: 3 + 4 = 7
Job = ["B", 3, 4] completed
Parallel task C: 5 + 6 = 11
Job = ["C", 5, 6] completed
```

## 1.25 Chapter 23: Esoteric: Writing custom functions to decide which jobs are up to date with `@check_if_upToDate`

See also:

- *Manual Table of Contents*
- *`@check_if_upToDate` syntax in detail*

### 1.25.1 `@check_if_upToDate` : Manual dependency checking

tasks specified with most decorators such as

- `@split`
- `@transform`
- `@merge`
- `@collate`
- `@collate`

have automatic dependency checking based on file modification times.

Sometimes, you might want to decide have more control over whether to run jobs, especially if a task does not rely on or produce files (i.e. with `@parallel`)

You can write your own custom function to decide whether to run a job. This takes as many parameters as your task function, and needs to return a tuple for whether an update is required, and why (i.e. `tuple(bool, str)`)

This simple example which creates the file "a.1" if it does not exist:

```
from ruffus import *
@originate("a.1")
def create_if_necessary(output_file):
    open(output_file, "w")

pipeline_run([])
```

could be rewritten more laboriously as:

```
from ruffus import *
import os
def check_file_exists(input_file, output_file):
    if os.path.exists(output_file):
        return False, "File already exists"
    return True, "%s is missing" % output_file
```

```

@parallel([None, "a.1"])
@check_if_upToDate(check_file_exists)
def create_if_necessary(input_file, output_file):
    open(output_file, "w")

pipeline_run([create_if_necessary])

```

**Both produce the same output:**

```

Task = create_if_necessary
Job = [null, "a.1"] completed

```

---

**Note:** The function specified by `@check_if_update` can be called more than once for each job.

See the *description here* of how *Ruffus* decides which tasks to run.

---

## 1.26 Appendix 1: Flow Chart Colours with `pipeline_printout_graph(...)`

See also:

- *Manual Table of Contents*
- `pipeline_printout_graph(...)`
- Download code
- *Code for experimenting with colours*

### 1.26.1 Flowchart colours

The appearance of *Ruffus* flowcharts produced by `pipeline_printout_graph` can be extensively customised.

This is mainly controlled by the `user_colour_scheme` (note UK spelling of “colour”) parameter

Example:

Use colour scheme index = 1

```

pipeline_printout_graph ("flowchart.svg", "svg", [final_task],
                        user_colour_scheme = {
                            "colour_scheme_index" : 1,
                            "Pipeline"          : {"fontcolor" : '#FF3232' },
                            "Key"               : {"fontcolor" : "Red",
                                      "fillcolor" : "#F6F4F4" },
                            "Task to run"       : {"linecolor" : "#0044A0" },
                            "Final target"     : {"fillcolor" : "#EFA03B",
                                      "fontcolor" : "black",
                                      "dashed"   : 0
                                    }
                        })

```

There are 8 colour schemes by setting “`colour_scheme_index`”:

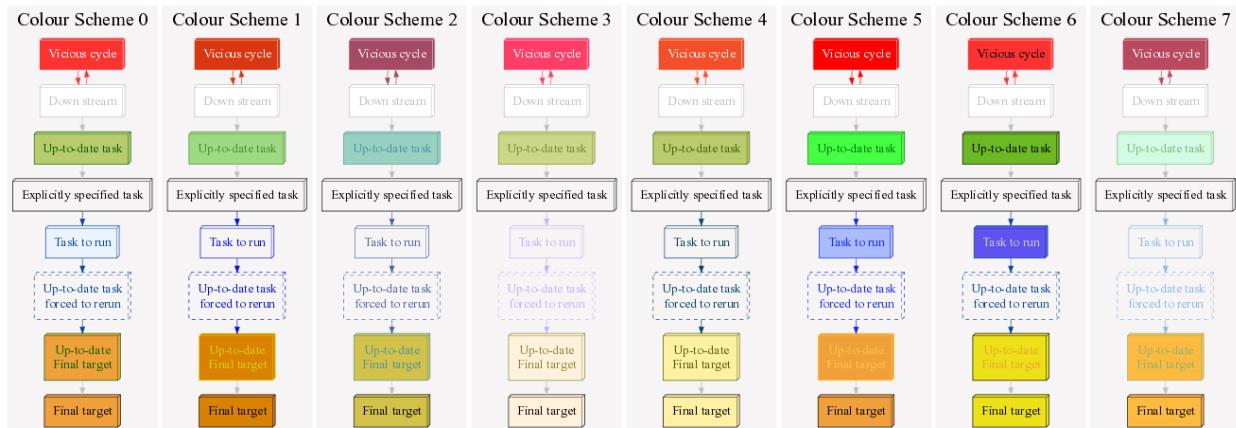
```

pipeline_printout_graph ("flowchart.svg", "svg", [final_task],
                        user_colour_scheme = {"colour_scheme_index" : 6})

```

These colours were chosen after much fierce arguments between the authors and friends, and much inspiration from <http://kuler.adobe.com/#create/fromacolor>. Please feel free to submit any additional sets of colours for our consideration.

(Click here for image in svg.)



## 1.27 Appendix 2: How dependency is checked

See also:

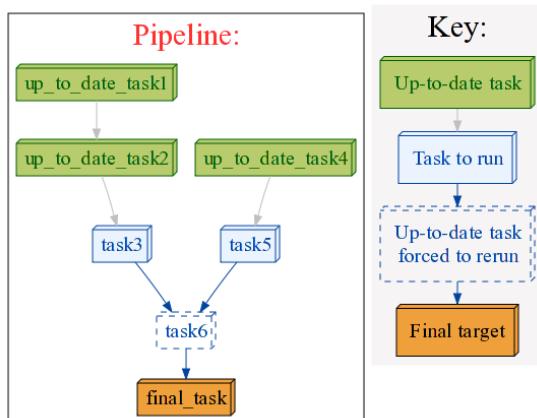
- *Manual Table of Contents*

### 1.27.1 Overview

How does *Ruffus* decide how to run your pipeline?

- In which order should pipelined functions be called?
- Which parts of the pipeline are up-to-date and do not need to be rerun?

#### Running all out-of-date tasks and dependents



By default, *Ruffus* will

- build a flow chart (dependency tree) of pipelined tasks (functions)
- start from the most ancestral tasks with the fewest dependencies (`task1` and `task4` in the flowchart above).
- walk up the tree to find the first incomplete / out-of-date tasks (i.e. `task3` and `task5`).
- start running from there

**All down-stream (dependent) tasks will be re-run anyway, so we don't have to test** whether they are up-to-date or not.

---

**Note:** This means that *Ruffus* may ask any task if their jobs are out of date more than once:

- once when deciding which parts of the pipeline have to be run
  - once just before executing the task.
- 

*Ruffus* tries to be clever / efficient, and does the minimal amount of querying.

## Forced Reruns

Even if a pipeline stage appears to be up to date, you can always force the pipeline to include from one or more task functions.

This is particularly useful, for example, if the pipeline data hasn't changed but the analysis or computational code has.

```
pipeline_run(forcedtorun_tasks = [up_to_date_task1])
```

will run all tasks from `up_to_date_task1` to `final_task`

Both the "target" and the "forced" lists can include as many tasks as you wish. All dependencies are still carried out and out-of-date jobs rerun.

## Esoteric option: Minimal Reruns

In the above example, if we were to delete the results of `up_to_date_task1`, *Ruffus* would rerun `up_to_date_task1`, `up_to_date_task2` and `task3`.

However, you might argue that so long as `up_to_date_task2` is up-to-date, and it is the only necessary prerequisite for `task3`, we should not be concerned about `up_to_date_task1`.

This is enabled with:

```
pipeline_run([task6], gnu_make_maximal_rebuild_mode = False)
```

This option walks down the dependency tree and proceeds no further when it encounters an up-to-date task (`up_to_date_task2`) whatever the state of what lies beyond it.

This rather dangerous option is useful if you don't want to keep all the intermediate files/results from upstream tasks. The pipeline will only not involve any incomplete tasks which precede an up-to-date result.

This is seldom what you intend, and you should always check that the appropriate stages of the pipeline are executed in the flowchart output.

## 1.28 Appendix 3: Exceptions thrown inside pipelines

### 1.28.1 Overview

The goal for *Ruffus* is that exceptions should just work *out-of-the-box* without any fuss. This is especially important for exceptions that come from your code which may be raised in a different process. Often multiple parallel operations (jobs or tasks) fail at the same time. *Ruffus* will forward each of these exceptions with the tracebacks so you can jump straight to the offending line.

This example shows separate exceptions from two jobs running in parallel:

```
from ruffus import *

@originate(["a.start", "b.start", "c.start", "d.start", "e.start"])
def throw_exceptions_here(output_file):
    raise Exception("OOPS")

pipeline_run(multiprocess = 2)

>>> pipeline_run(multiprocess = 2)

ruffus.ruffus_exceptions.RethrownJobError:

Original exceptions:

Exception #1
'exceptions.Exception(OOPS)' raised in ...
Task = def throw_exceptions_here(...):
Job   = [None -> b.start]

Traceback (most recent call last):
File "/usr/local/lib/python2.7/dist-packages/ruffus/task.py", line 685, in run_pooled_
    return_value = job_wrapper(param, user_defined_work_func, register_cleanup, touch_f
File "/usr/local/lib/python2.7/dist-packages/ruffus/task.py", line 549, in job_wrapper
    job_wrapper_io_files(param, user_defined_work_func, register_cleanup, touch_files_on_
File "/usr/local/lib/python2.7/dist-packages/ruffus/task.py", line 504, in job_wrapper
    ret_val = user_defined_work_func(*(param[1:]))
File "<stdin>", line 3, in throw_exceptions_here
Exception: OOPS

Exception #2
'exceptions.Exception(OOPS)' raised in ...
Task = def throw_exceptions_here(...):
Job   = [None -> a.start]

Traceback (most recent call last):
File "/usr/local/lib/python2.7/dist-packages/ruffus/task.py", line 685, in run_pooled_
    return_value = job_wrapper(param, user_defined_work_func, register_cleanup, touch_f
File "/usr/local/lib/python2.7/dist-packages/ruffus/task.py", line 549, in job_wrapper
    job_wrapper_io_files(param, user_defined_work_func, register_cleanup, touch_files_on_
File "/usr/local/lib/python2.7/dist-packages/ruffus/task.py", line 504, in job_wrapper
    ret_val = user_defined_work_func(*(param[1:]))
File "<stdin>", line 3, in throw_exceptions_here
Exception: OOPS

.. image:: ../../images/manual_exceptions.png
```

## 1.28.2 Pipelines running in parallel accumulate Exceptions

As show above, by default *Ruffus* accumulates NN exceptions before interrupting the pipeline prematurely where NN is the specified parallelism for `pipeline_run(multiprocess = NN)`

This seems a fair tradeoff between being able to gather detailed error information for running jobs, and not wasting too much time for a task that is going to fail anyway.

## 1.28.3 Terminate pipeline immediately upon Exceptions

Set `pipeline_run(exceptions_terminate_immediately = True)`

To have all exceptions interrupt the pipeline immediately, invoke:

```
pipeline_run(exceptions_terminate_immediately = True)
```

For example, with this change, only a single exception will be thrown before the pipeline is interrupted:

```
from ruffus import *

@originate(["a.start", "b.start", "c.start", "d.start", "e.start"])
def throw_exceptions_here(output_file):
    raise Exception("OOPS")

pipeline_run(multiprocess = 2, exceptions_terminate_immediately = True)

>>> pipeline_run(multiprocess = 2)
ruffus.ruffus_exceptions.RethrownJobError:

Original exception:

Exception #1
'exceptions.Exception(OOPS)' raised in ...
Task = def throw_exceptions_here(...):
Job   = [None -> a.start]

Traceback (most recent call last):
[Tedious traceback snipped out!!!!....]
Exception: OOPS
```

`raise Ruffus.JobSignalledBreak`

The same can be accomplished on a finer scale by throwing the `Ruffus.JobSignalledBreak` Exception. Unlike other exceptions, this causes an immediate halt in pipeline execution. If there are other exceptions in play at that point, they will be rethrown in the main process but no new exceptions will be added.

```
from ruffus import *

@originate(["a.start", "b.start", "c.start", "d.start", "e.start"])
def throw_exceptions_here(output_file):
```

```
    raise JobSignalledBreak("OOPS")

pipeline_run(multiprocess = 2)
```

## 1.28.4 Display exceptions as they occur

In the following example, the jobs throw exceptions at two second staggered intervals into the job. With `log_exceptions = True`, the exceptions are displayed as they occur even though the pipeline continues running.

`logger.error(...)` will be invoked with the string representation of the each exception, and associated stack trace.

The default logger prints to `sys.stderr`, but as usual can be changed to any class from the logging module or compatible object via `pipeline_run(logger = XXX)`

```
from ruffus import *
import time, os

@originate(["1.start", "2.start", "3.start", "4.start", "5.start"])
def throw_exceptions_here(output_file):
    delay = int(os.path.splitext(output_file)[0])
    time.sleep(delay * 2)
    raise JobSignalledBreak("OOPS")

pipeline_run(log_exceptions = True, multiprocess = 5)
```

## 1.29 Appendix 4: Names exported from Ruffus

See also:

- *Manual Table of Contents*

### 1.29.1 Ruffus Names

This is a list of all the names *Ruffus* makes available:

Category	Manual
<b>Pipeline functions</b>	<p><i>pipeline_printout() (Manual)</i>  <i>pipeline_printout() (Manual)</i>  <i>pipeline_printout() (Manual)</i></p>
<b>Decorators</b>	<p><i>@active_if (Manual)</i>  <i>@check_if_uptodate (Manual)</i>  <i>@collate (Manual)</i>  <i>@files (Manual)</i>  <i>@follows (Manual)</i>  <i>@jobs_limit (Manual)</i>  <i>@merge (Manual)</i>  <i>@mkdir (Manual)</i>  <i>@originate (Manual)</i>  <i>@parallel (Manual)</i>  <i>@posttask (Manual)</i>  <i>@split (Manual)</i>  <i>@subdivide (Manual)</i>  <i>@transform (Manual)</i>  <i>@files_re (Manual)</i></p>
<b>Loggers</b>	<p>stderr_logger  black_hole_logger</p>
<b>Parameter disambiguating Indicators</b>	<p><i>suffix (Manual)</i>  <i>regex (Manual)</i>  <i>formatter (Manual)</i>  <i>inputs (Manual)</i>  <i>inputs (Manual)</i>  <i>touch_file (Manual)</i>  <i>combine</i>  <i>mkdir (Manual)</i>  <i>output_from (Manual)</i></p>
<b>Decorators in ruffus.combinatorics</b>	<p><i>@combinations (Manual)</i>  <i>@combinations_with_replacement (Manual)</i>  <i>@permutations (Manual)</i>  <i>@product (Manual)</i></p>
<b>Decorators in ruffus.cmdline</b>	<p><i>get_argparse</i>  <i>setup_logging</i>  <i>run</i></p>
<b>1.29. Appendix 4: Names exported from Ruffus</b>	<b>MESSAGE</b>

## 1.30 Appendix 5: @files: Deprecated syntax

**Warning:**

- This is deprecated syntax  
which is no longer supported and  
should NOT be used in new code.

**See also:**

- *Manual Table of Contents*
- *decorators*
- @files syntax in detail

### 1.30.1 Overview

The python functions which do the actual work of each stage or *task* of a *Ruffus* pipeline are written by you.

The role of *Ruffus* is to make sure these functions are called in the right order, with the right parameters, running in parallel using multiprocessing if desired.

The easiest way to specify parameters to *Ruffus task* functions is to use the @files decorator.

### 1.30.2 @files

Running this code:

```
from ruffus import *

@files('a.1', ['a.2', 'b.2'], 'A file')
def single_job_io_task(infile, outfiles, text):
    for o in outfiles: open(o, "w")

# prepare input file
open('a.1', "w")

pipeline_run()
```

Is equivalent to calling:

```
single_job_io_task('a.1', ['a.2', 'b.2'], 'A file')
```

And produces:

```
>>> pipeline_run()
Job = [a.1 -> [a.2, b.2], A file] completed
Completed Task = single_job_io_task
```

*Ruffus* will automatically check if your task is up to date. The second time *pipeline\_run()* is called, nothing will happen. But if you update *a.1*, the task will rerun:

```
>>> open('a.1', "w")
>>> pipeline_run()
Job = [a.1 -> [a.2, b.2], A file] completed
Completed Task = single_job_io_task
```

See *chapter 2* for a more in-depth discussion of how *Ruffus* decides which parts of the pipeline are complete and up-to-date.

### 1.30.3 Running the same code on different parameters in parallel

Your pipeline may require the same function to be called multiple times on independent parameters. In which case, you can supply all the parameters to @files, each will be sent to separate jobs that may run in parallel if necessary. *Ruffus* will check if each separate job is up-to-date using the *inputs* and *outputs* (first two) parameters (See the *Up-to-date jobs are not re-run unnecessarily* ).

For example, if a sequence (e.g. a list or tuple) of 5 parameters are passed to @files, that indicates there will also be 5 separate jobs:

```
from ruffus import *
parameters = [
    [ 'job1.file' , ] , # 1st job
    [ 'job2.file' , 4 ] , # 2st job
    [ 'job3.file' , [3, 2] ] , # 3st job
    [ 67, [13, 'job4.file'] ] , # 4st job
    [ 'job5.file' ] , # 5st job
]
@files(parameters)
def task_file(*params):
    ""
```

*Ruffus* creates as many jobs as there are elements in parameters.

In turn, each of these elements consist of series of parameters which will be passed to each separate job.

Thus the above code is equivalent to calling:

```
task_file('job1.file')
task_file('job2.file', 4)
task_file('job3.file', [3, 2])
task_file(67, [13, 'job4.file'])
task_file('job5.file')
```

What `task_file()` does with these parameters is up to you!

The only constraint on the parameters is that *Ruffus* will treat any first parameter of each job as the *inputs* and any second as the *output*. Any strings in the *inputs* or *output* parameters (including those nested in sequences) will be treated as file names.

Thus, to pick the parameters out of one of the above jobs:

```
task_file(67, [13, 'job4.file'])
```

```
inputs == 67
outputs == [13, 'job4.file']
```

The solitary output filename is job4.file

## Checking if jobs are up to date

Usually we do not want to run all the stages in a pipeline but only where the input data has changed or is no longer up to date.

One easy way to do this is to check the modification times for files produced at each stage of the pipeline.

Let us first create our starting files a.1 and b.1

We can then run the following pipeline function to create

- a.2 from a.1 and
- b.2 from b.1

```
# create starting files
open("a.1", "w")
open("b.1", "w")

from ruffus import *
parameters = [
    [ 'a.1', 'a.2', 'A file'], # 1st job
    [ 'b.1', 'b.2', 'B file'], # 2nd job
]

@files(parameters)
def parallel_io_task(infile, outfile, text):
    # copy infile contents to outfile
    infile_text = open(infile).read()
    f = open(outfile, "w").write(infile_text + "\n" + text)

pipeline_run()
```

This produces the following output:

```
>>> pipeline_run()
Job = [a.1 -> a.2, A file] completed
Job = [b.1 -> b.2, B file] completed
Completed Task = parallel_io_task
```

If you called `pipeline_run()` again, nothing would happen because the files are up to date:

a.2 is more recent than a.1 and  
b.2 is more recent than b.1

**However, if you subsequently modified a.1 again:**

```
open("a.1", "w")
pipeline_run(verbose = 1)
```

you would see the following:

```
>>> pipeline_run([parallel_io_task])
Task = parallel_io_task
    Job = ["a.1" -> "a.2", "A file"] completed
    Job = ["b.1" -> "b.2", "B file"] unnecessary: already up to date
Completed Task = parallel_io_task
```

The 2nd job is up to date and will be skipped.

## 1.31 Appendix 6: @files\_re: Deprecated syntax using regular expressions

**Warning:**

- This is deprecated syntax  
which is no longer supported and  
should NOT be used in new code.

See also:

- *Manual Table of Contents*
- *decorators*
- @files\_re syntax in detail

### 1.31.1 Overview

@files\_re combines the functionality of @transform, @collate and @merge in one overloaded decorator.

This is the reason why its use is discouraged. @files\_re syntax is far too overloaded and context-dependent to support its myriad of different functions.

The following documentation is provided to help maintain historical *Ruffus* usage.

#### Transforming input and output filenames

For example, the following code takes files from the previous pipeline task, and makes new output parameters with the .sums suffix in place of the .chunks suffix:

```
@transform(step_4_split_numbers_into_chunks, suffix(".chunks"), ".sums")
def step_5_calculate_sum_of_squares (input_file_name, output_file_name):
    """
        calculate sums and sums of squares for all values in the input_file_name
        writing to output_file_name
    """
```

This can be written using @files\_re equivalently:

```
@files_re(step_4_split_numbers_into_chunks, r".chunks", r".sums")
def step_5_calculate_sum_of_squares (input_file_name, output_file_name):
    """
```

## Collating many *inputs* into a single *output*

Similarly, the following code collects **inputs** from the same species in the same directory:

```
@collate('*.animals',
          regex(r'mammals.([^.]+)'),
          r'\1/animals.in_my_zoo',
          r'\1' )
def capture_mammals(infiles, outfile, species):
    # summarise all animals of this species
    """
```

This can be written using @files\_re equivalently using the *combine* indicator:

```
@files_re('*.animals',
           regex(r'mammals.([^.]+)'),
           combine(r'\1/animals.in_my_zoo'),
           r'\1' )
def capture_mammals(infiles, outfile, species):
    # summarise all animals of this species
    """
```

## Generating *input* and *output* parameter using regular expressions

The following code generates additional *input* prerequisite file names which match the original *input* files.

We want each job of our analyse() function to get corresponding pairs of xx.chunks and xx.red\_indian files when

```
*.chunks are generated by the task function split_up_problem() and
*.red_indian are generated by the task function make_red_indians():

@follows(make_red_indians)
@transform(split_up_problem,
          regex(r"(.*)chunks"),
          inputs([r"\g<0>",
                  r"\1.red_indian"]),
          r"\1.results"
          )
def analyse(input_filenames, output_file_name):
    "Do analysis here"
```

The equivalent code using @files\_re looks very similar:

```
@follows(make_red_indians)
@files_re( split_up_problem,
           regex(r"(.*)chunks"),
           [r"\g<0>",
            r"\1.red_indian"]),
           r"\1.results"
def analyse(input_filenames, output_file_name):
    "Do analysis here"
```

Example code for:

## 1.32 Chapter 1: Python Code for An introduction to basic Ruffus syntax

See also:

- [Manual Table of Contents](#)
- [@transform syntax in detail](#)
- Back to [Chapter 1: An introduction to basic Ruffus syntax](#)

### 1.32.1 Your first Ruffus script

```
::

#
# The starting data files would normally exist beforehand!
# We create some empty files for this example
#
starting_files = ["a.fasta", "b.fasta", "c.fasta"]

for ff in starting_files:
    open(ff, "w")

from ruffus import *

#
# STAGE 1 fasta->sam
#
@transform(starting_files,
           suffix(".fasta"),
           ".sam")
def map_dna_sequence(input_file,
                      output_file):
    ii = open(input_file)
    oo = open(output_file, "w")

#
# STAGE 2 sam->bam
#
@transform(map_dna_sequence,
           suffix(".sam"),
           ".bam")
def compress_sam_file(input_file,
                      output_file):
    ii = open(input_file)
    oo = open(output_file, "w")

#
# STAGE 3 bam->statistics
#
@transform(compress_sam_file,
           suffix(".bam"),
           ".statistics",
           "use_linear_model")
def summarise_bam_file(input_file,
```

```
        output_file,
        extra_stats_parameter):
"""
Sketch of real analysis function
"""
ii = open(input_file)
oo = open(output_file, "w")

pipeline_run()
```

### 1.32.2 Resulting Output

```
>>> pipeline_run()
Job   = [a.fasta -> a.sam] completed
Job   = [b.fasta -> b.sam] completed
Job   = [c.fasta -> c.sam] completed
Completed Task = map_dna_sequence
    Job   = [a.sam -> a.bam] completed
    Job   = [b.sam -> b.bam] completed
    Job   = [c.sam -> c.bam] completed
Completed Task = compress_sam_file
    Job   = [a.bam -> a.statistics, use_linear_model] completed
    Job   = [b.bam -> b.statistics, use_linear_model] completed
    Job   = [c.bam -> c.statistics, use_linear_model] completed
Completed Task = summarise_bam_file
```

## 1.33 Chapter 1: Python Code for Transforming data in a pipeline with `@transform`

See also:

- *Manual Table of Contents*
- *@transform syntax in detail*
- Back to **Chapter 2: Transforming data in a pipeline with @transform**

### 1.33.1 Your first Ruffus script

```
#  
#   The starting data files would normally exist beforehand!  
#       We create some empty files for this example  
#  
starting_files = [ ("a.1.fastq", "a.2.fastq"),  
                  ("b.1.fastq", "b.2.fastq"),  
                  ("c.1.fastq", "c.2.fastq")]  
  
for ff_pair in starting_files:  
    open(ff_pair[0], "w")  
    open(ff_pair[1], "w")
```

```

from ruffus import *

#
# STAGE 1 fasta->sam
#
@transform(starting_files,
           suffix(".1.fastq"),
           ".sam")
def map_dna_sequence(input_files,
                      output_file):
    # remember there are two input files now
    ii1 = open(input_files[0])
    ii2 = open(input_files[1])
    oo = open(output_file, "w")

#
# STAGE 2 sam->bam
#
@transform(map_dna_sequence,
           suffix(".sam"),
           ".bam")
def compress_sam_file(input_file,
                      output_file):
    ii = open(input_file)
    oo = open(output_file, "w")

#
# STAGE 3 bam->statistics
#
@transform(compress_sam_file,
           suffix(".bam"),
           ".statistics",
           "use_linear_model")
def summarise_bam_file(input_file,
                        output_file,
                        extra_stats_parameter):
    """
    Sketch of real analysis function
    """
    ii = open(input_file)
    oo = open(output_file, "w")

pipeline_run()

```

### 1.33.2 Resulting Output

```

>>> pipeline_run()
Job   = [[a.1.fastq, a.2.fastq] -> a.sam] completed
Job   = [[b.1.fastq, b.2.fastq] -> b.sam] completed
Job   = [[c.1.fastq, c.2.fastq] -> c.sam] completed
Completed Task = map_dna_sequence
Job   = [a.sam -> a.bam] completed
Job   = [b.sam -> b.bam] completed
Job   = [c.sam -> c.bam] completed
Completed Task = compress_sam_file
Job   = [a.bam -> a.statistics, use_linear_model] completed

```

```
Job   = [b.bam -> b.statistics, use_linear_model] completed
Job   = [c.bam -> c.statistics, use_linear_model] completed
Completed Task = summarise_bam_file
```

## 1.34 Chapter 3: Python Code for More on @transform-ing data

See also:

- *Manual Table of Contents*
- *@transform syntax in detail*
- Back to **Chapter 3: More on @transform-ing data and @originate**

### 1.34.1 Producing several items / files per job

```
from ruffus import *

#-----
# Create pairs of input files
#
first_task_params = [
    ['job1.a.start', 'job1.b.start'],
    ['job2.a.start', 'job2.b.start'],
    ['job3.a.start', 'job3.b.start'],
]

for input_file_pairs in first_task_params:
    for input_file in input_file_pairs:
        open(input_file, "w")

#-----
# first task
#
@transform(first_task_params, suffix(".start"),
          [".output.1",
           ".output.extra.1"],
          "some_extra.string.for_example", 14)
def first_task(input_files, output_file_pair,
               extra_parameter_str, extra_parameter_num):
    for output_file in output_file_pair:
        with open(output_file, "w"):
            pass

#-----
# second task
#
@transform(first_task, suffix(".output.1"), ".output2")
def second_task(input_files, output_file):
    with open(output_file, "w"): pass
```

```
#-----
#
#      Run
#
# pipeline_run([second_task])
```

## Resulting Output

```
>>> pipeline_run([second_task])
Job  = [[job1.a.start, job1.b.start] -> [job1.a.output.1, job1.a.output.extra.1], some_extra
Job  = [[job2.a.start, job2.b.start] -> [job2.a.output.1, job2.a.output.extra.1], some_extra
Job  = [[job3.a.start, job3.b.start] -> [job3.a.output.1, job3.a.output.extra.1], some_extra
Completed Task = first_task
    Job = [[job1.a.output.1, job1.a.output.extra.1] -> job1.a.output2] completed
    Job = [[job2.a.output.1, job2.a.output.extra.1] -> job2.a.output2] completed
    Job = [[job3.a.output.1, job3.a.output.extra.1] -> job3.a.output2] completed
Completed Task = second_task
```

### 1.34.2 Defining tasks function out of order

```
from ruffus import *

#-----
#   Create pairs of input files
#
first_task_params = [
    ['job1.a.start', 'job1.b.start'],
    ['job2.a.start', 'job2.b.start'],
    ['job3.a.start', 'job3.b.start'],
]

for input_file_pairs in first_task_params:
    for input_file in input_file_pairs:
        open(input_file, "w")

#-----
#   second task defined first
#
#   task name string wrapped in output_from(...)
@transform(output_from("first_task"), suffix(".output.1"), ".output2")
def second_task(input_files, output_file):
    with open(output_file, "w"): pass

#-----
#   first task
#
@transform(first_task_params, suffix(".start"),
          [".output.1",
           ".output.extra.1"],
```

```
        "some_extra.string.for_example", 14)
def first_task(input_files, output_file_pair,
               extra_parameter_str, extra_parameter_num):
    for output_file in output_file_pair:
        with open(output_file, "w"):
            pass

#-----
#
#      Run
#
# pipeline_run([second_task])
```

## Resulting Output

```
>>> pipeline_run([second_task])
Job   = [[job1.a.start, job1.b.start] -> [job1.a.output.1, job1.a.output.extra.1], some_extra
Job   = [[job2.a.start, job2.b.start] -> [job2.a.output.1, job2.a.output.extra.1], some_extra
Job   = [[job3.a.start, job3.b.start] -> [job3.a.output.1, job3.a.output.extra.1], some_extra
Completed Task = first_task
    Job   = [[job1.a.output.1, job1.a.output.extra.1] -> job1.a.output2] completed
    Job   = [[job2.a.output.1, job2.a.output.extra.1] -> job2.a.output2] completed
    Job   = [[job3.a.output.1, job3.a.output.extra.1] -> job3.a.output2] completed
Completed Task = second_task
```

### 1.34.3 Multiple dependencies

```
from ruffus import *
import time
import random

#-----
#  Create pairs of input files
#
first_task_params = [
    ['job1.a.start', 'job1.b.start'],
    ['job2.a.start', 'job2.b.start'],
    ['job3.a.start', 'job3.b.start'],
]
second_task_params = [
    ['job4.a.start', 'job4.b.start'],
    ['job5.a.start', 'job5.b.start'],
    ['job6.a.start', 'job6.b.start'],
]

for input_file_pairs in first_task_params + second_task_params:
    for input_file in input_file_pairs:
        open(input_file, "w")

#-----
#
#  first task
```

```

#
@transform(first_task_params, suffix(".start"),
           [".output.1",
            ".output.extra.1"],
           "some_extra.string.for_example", 14)
def first_task(input_files, output_file_pair,
               extra_parameter_str, extra_parameter_num):
    for output_file in output_file_pair:
        with open(output_file, "w"):
            pass
    time.sleep(random.random())

#-----
#
#   second task
#
@transform(second_task_params, suffix(".start"),
           [".output.1",
            ".output.extra.1"],
           "some_extra.string.for_example", 14)
def second_task(input_files, output_file_pair,
                extra_parameter_str, extra_parameter_num):
    for output_file in output_file_pair:
        with open(output_file, "w"):
            pass
    time.sleep(random.random())

#-----
#
#   third task
#
#       depends on both first_task() and second_task()
@transform([first_task, second_task], suffix(".output.1"), ".output2")
def third_task(input_files, output_file):
    with open(output_file, "w"): pass

#-----
#
#       Run
#
pipeline_run([third_task], multiprocess = 6)

```

## Resulting Output

```

>>> pipeline_run([third_task], multiprocess = 6)
Job  = [[job3.a.start, job3.b.start] -> [job3.a.output.1, job3.a.output.extra.1], some_extra
Job  = [[job6.a.start, job6.b.start] -> [job6.a.output.1, job6.a.output.extra.1], some_extra
Job  = [[job1.a.start, job1.b.start] -> [job1.a.output.1, job1.a.output.extra.1], some_extra
Job  = [[job4.a.start, job4.b.start] -> [job4.a.output.1, job4.a.output.extra.1], some_extra
Job  = [[job5.a.start, job5.b.start] -> [job5.a.output.1, job5.a.output.extra.1], some_extra
Completed Task = second_task
Job  = [[job2.a.start, job2.b.start] -> [job2.a.output.1, job2.a.output.extra.1], some_extra
Completed Task = first_task

```

```
Job = [[job1.a.output.1, job1.a.output.extra.1] -> job1.a.output2] completed
Job = [[job2.a.output.1, job2.a.output.extra.1] -> job2.a.output2] completed
Job = [[job3.a.output.1, job3.a.output.extra.1] -> job3.a.output2] completed
Job = [[job4.a.output.1, job4.a.output.extra.1] -> job4.a.output2] completed
Job = [[job5.a.output.1, job5.a.output.extra.1] -> job5.a.output2] completed
Job = [[job6.a.output.1, job6.a.output.extra.1] -> job6.a.output2] completed
Completed Task = third_task
```

#### 1.34.4 Multiple dependencies after @follows

```
from ruffus import *
import time
import random

#-----
# Create pairs of input files
#
first_task_params = [
    ['job1.a.start', 'job1.b.start'],
    ['job2.a.start', 'job2.b.start'],
    ['job3.a.start', 'job3.b.start'],
]
second_task_params = [
    ['job4.a.start', 'job4.b.start'],
    ['job5.a.start', 'job5.b.start'],
    ['job6.a.start', 'job6.b.start'],
]

for input_file_pairs in first_task_params + second_task_params:
    for input_file in input_file_pairs:
        open(input_file, "w")

#-----
# first task
#
@transform(first_task_params, suffix(".start"),
          [".output.1",
           ".output.extra.1"],
          "some_extra.string.for_example", 14)
def first_task(input_files, output_file_pair,
               extra_parameter_str, extra_parameter_num):
    for output_file in output_file_pair:
        with open(output_file, "w"):
            pass
    time.sleep(random.random())

#-----
# second task
#
@follows("first_task")
```

```

@transform(second_task_params, suffix(".start"),
          [".output.1",
           ".output.extra.1"],
          "some_extra.string.for_example", 14)
def second_task(input_files, output_file_pair,
                extra_parameter_str, extra_parameter_num):
    for output_file in output_file_pair:
        with open(output_file, "w"):
            pass
    time.sleep(random.random())

#-----
#
#      third task
#
#      depends on both first_task() and second_task()
@transform([first_task, second_task], suffix(".output.1"), ".output2")
def third_task(input_files, output_file):
    with open(output_file, "w"): pass

#-----
#
#      Run
#
# pipeline_run([third_task], multiprocess = 6)

```

### Resulting Output: `first_task` completes before `second_task`

```

>>> pipeline_run([third_task], multiprocess = 6)
Job  = [[job2.a.start, job2.b.start] -> [job2.a.output.1, job2.a.output.extra.1], some_extra
Job  = [[job3.a.start, job3.b.start] -> [job3.a.output.1, job3.a.output.extra.1], some_extra
Job  = [[job1.a.start, job1.b.start] -> [job1.a.output.1, job1.a.output.extra.1], some_extra
Completed Task = first_task
Job  = [[job4.a.start, job4.b.start] -> [job4.a.output.1, job4.a.output.extra.1], some_extra
Job  = [[job6.a.start, job6.b.start] -> [job6.a.output.1, job6.a.output.extra.1], some_extra
Job  = [[job5.a.start, job5.b.start] -> [job5.a.output.1, job5.a.output.extra.1], some_extra
Completed Task = second_task
Job  = [[job1.a.output.1, job1.a.output.extra.1] -> job1.a.output2] completed
Job  = [[job2.a.output.1, job2.a.output.extra.1] -> job2.a.output2] completed
Job  = [[job3.a.output.1, job3.a.output.extra.1] -> job3.a.output2] completed
Job  = [[job4.a.output.1, job4.a.output.extra.1] -> job4.a.output2] completed
Job  = [[job5.a.output.1, job5.a.output.extra.1] -> job5.a.output2] completed
Job  = [[job6.a.output.1, job6.a.output.extra.1] -> job6.a.output2] completed

```

## 1.35 Chapter 4: Python Code for Creating files with @originate

### See also:

- [Manual Table of Contents](#)
- [@transform syntax in detail](#)
- [Back to Chapter 4: @originate](#)

### 1.35.1 Using @originate

```
from ruffus import *

#-----
#  create initial files
#
@originate([  ['job1.a.start', 'job1.b.start'],
              ['job2.a.start', 'job2.b.start'],
              ['job3.a.start', 'job3.b.start']      ])
def create_initial_file_pairs(output_files):
    # create both files as necessary
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

#-----
#  first task
@transform(create_initial_file_pairs, suffix(".start"), ".output.1")
def first_task(input_files, output_file):
    with open(output_file, "w"): pass

#-----
#  second task
@transform(first_task, suffix(".output.1"), ".output.2")
def second_task(input_files, output_file):
    with open(output_file, "w"): pass

#
#      Run
#
pipeline_run([second_task])
```

### 1.35.2 Resulting Output

```
Job  = [None -> [job1.a.start, job1.b.start]] completed
Job  = [None -> [job2.a.start, job2.b.start]] completed
Job  = [None -> [job3.a.start, job3.b.start]] completed
Completed Task = create_initial_file_pairs
    Job  = [[job1.a.start, job1.b.start] -> job1.a.output.1] completed
    Job  = [[job2.a.start, job2.b.start] -> job2.a.output.1] completed
    Job  = [[job3.a.start, job3.b.start] -> job3.a.output.1] completed
Completed Task = first_task
    Job  = [job1.a.output.1 -> job1.a.output.2] completed
    Job  = [job2.a.output.1 -> job2.a.output.2] completed
    Job  = [job3.a.output.1 -> job3.a.output.2] completed
Completed Task = second_task
```

## 1.36 Chapter 5: Python Code for Understanding how your pipeline works with *pipeline\_printout(...)*

See also:

- *Manual Table of Contents*

- `pipeline_printout(...)` syntax
- Back to Chapter 5: *Understanding how your pipeline works*

### 1.36.1 Display the initial state of the pipeline

```
from ruffus import *
import sys

#-----
#   create initial files
#
@originate([
    ['job1.a.start', 'job1.b.start'],
    ['job2.a.start', 'job2.b.start'],
    ['job3.a.start', 'job3.b.start']    ])
def create_initial_file_pairs(output_files):
    # create both files as necessary
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

#-----
#   first task
@transform(create_initial_file_pairs, suffix(".start"), ".output.1")
def first_task(input_files, output_file):
    with open(output_file, "w"): pass

#-----
#   second task
@transform(first_task, suffix(".output.1"), ".output.2")
def second_task(input_files, output_file):
    with open(output_file, "w"): pass

pipeline_printout(sys.stdout, [second_task])
pipeline_printout(sys.stdout, [second_task], verbose = 3)
```

### 1.36.2 Normal Output

```
>>> pipeline_printout(sys.stdout, [second_task])
```

---

Tasks which will be run:

```
Task = create_initial_file_pairs
Task = first_task
Task = second_task
```

### 1.36.3 High Verbosity Output

```
>>> pipeline_printout(sys.stdout, [second_task], verbose = 3)
```

---

Tasks which will be run:

```

Task = create_initial_file_pairs
Job  = [None
       -> job1.a.start
       -> job1.b.start]
Job needs update: Missing files [job1.a.start, job1.b.start]
Job  = [None
       -> job2.a.start
       -> job2.b.start]
Job needs update: Missing files [job2.a.start, job2.b.start]
Job  = [None
       -> job3.a.start
       -> job3.b.start]
Job needs update: Missing files [job3.a.start, job3.b.start]

Task = first_task
Job  = [[job1.a.start, job1.b.start]
        -> job1.a.output.1]
Job needs update: Missing files [job1.a.start, job1.b.start, job1.a.output.1]
Job  = [[job2.a.start, job2.b.start]
        -> job2.a.output.1]
Job needs update: Missing files [job2.a.start, job2.b.start, job2.a.output.1]
Job  = [[job3.a.start, job3.b.start]
        -> job3.a.output.1]
Job needs update: Missing files [job3.a.start, job3.b.start, job3.a.output.1]

Task = second_task
Job  = [job1.a.output.1
       -> job1.a.output.2]
Job needs update: Missing files [job1.a.output.1, job1.a.output.2]
Job  = [job2.a.output.1
       -> job2.a.output.2]
Job needs update: Missing files [job2.a.output.1, job2.a.output.2]
Job  = [job3.a.output.1
       -> job3.a.output.2]
Job needs update: Missing files [job3.a.output.1, job3.a.output.2]

```

---

### 1.36.4 Display the partially up-to-date pipeline

Run the pipeline, modify `job1.stage` so that the second task is no longer up-to-date and printout the pipeline stage again:

```

>>> pipeline_run([second_task])
Job  = [None -> [job1.a.start, job1.b.start]] completed
Job  = [None -> [job2.a.start, job2.b.start]] completed
Job  = [None -> [job3.a.start, job3.b.start]] completed
Completed Task = create_initial_file_pairs
Job  = [[job1.a.start, job1.b.start] -> job1.a.output.1] completed
Job  = [[job2.a.start, job2.b.start] -> job2.a.output.1] completed
Job  = [[job3.a.start, job3.b.start] -> job3.a.output.1] completed
Completed Task = first_task
Job  = [job1.a.output.1 -> job1.a.output.2] completed
Job  = [job2.a.output.1 -> job2.a.output.2] completed
Job  = [job3.a.output.1 -> job3.a.output.2] completed
Completed Task = second_task

```

```
# modify job1.stage1
>>> open("job1.a.output.1", "w").close()
```

At a verbosity of 5, even jobs which are up-to-date will be displayed:

```
>>> pipeline_printout(sys.stdout, [second_task], verbose = 5)
```

---

Tasks which are up-to-date:

```
Task = create_initial_file_pairs
      Job = [None
              -> job1.a.start
              -> job1.b.start]
      Job up-to-date
      Job = [None
              -> job2.a.start
              -> job2.b.start]
      Job up-to-date
      Job = [None
              -> job3.a.start
              -> job3.b.start]
      Job up-to-date

Task = first_task
      Job = [[job1.a.start, job1.b.start]
              -> job1.a.output.1]
      Job up-to-date
      Job = [[job2.a.start, job2.b.start]
              -> job2.a.output.1]
      Job up-to-date
      Job = [[job3.a.start, job3.b.start]
              -> job3.a.output.1]
      Job up-to-date
```

---

Tasks which will be run:

```
Task = second_task
      Job = [job1.a.output.1
              -> job1.a.output.2]
      Job needs update:
      Input files:
          * 05 Dec 2013 12:04:52.80: job1.a.output.1
      Output files:
          * 05 Dec 2013 12:01:29.01: job1.a.output.2

      Job = [job2.a.output.1
              -> job2.a.output.2]
      Job up-to-date
      Job = [job3.a.output.1
              -> job3.a.output.2]
      Job up-to-date
```

---

We can now see that the there is only one job in “second\_task” which needs to be re-run because ‘job1.stage1’ has been modified after ‘job1.stage2’

## 1.37 Chapter 7: Python Code for Displaying the pipeline visually with *pipeline\_printout\_graph(...)*

See also:

- *Manual Table of Contents*
- *pipeline\_printout\_graph(...)* syntax
- Back to [Chapter 7: Displaying the pipeline visually](#)

### 1.37.1 Code

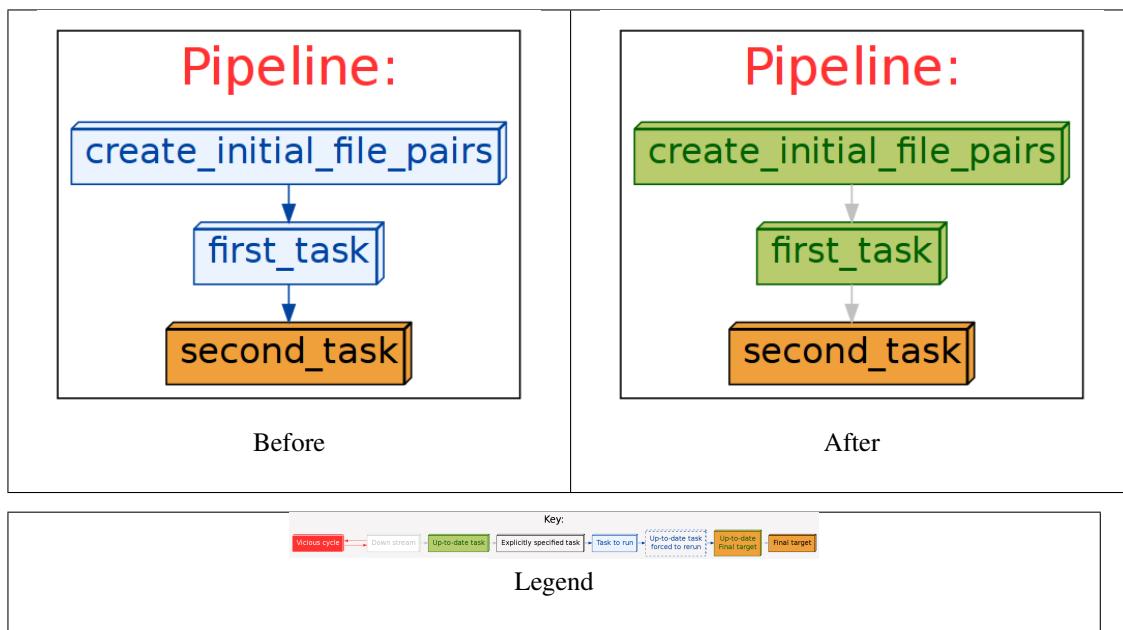
```
1  from ruffus import *
2  import sys
3
4  #-----
5  #      create initial files
6  #
7  @originate([
8      ['job1.a.start', 'job1.b.start'],
9      ['job2.a.start', 'job2.b.start'],
10     ['job3.a.start', 'job3.b.start']    ])
11 def create_initial_file_pairs(output_files):
12     # create both files as necessary
13     for output_file in output_files:
14         with open(output_file, "w") as oo: pass
15
16     #-----
17     #      first task
18     @transform(create_initial_file_pairs, suffix(".start"), ".output.1")
19     def first_task(input_files, output_file):
20         with open(output_file, "w"): pass
21
22     #-----
23     #      second task
24     @transform(first_task, suffix(".output.1"), ".output.2")
25     def second_task(input_files, output_file):
26         with open(output_file, "w"): pass
27
28     # Print graph before running pipeline
29
30     #-----
31     #
32     #      Show flow chart and tasks before running the pipeline
33     #
34     print "Show flow chart and tasks before running the pipeline"
35     pipeline_printout_graph ( open("simple_tutorial_stage5_before.png", "w"),
36                               "png",
37                               [second_task],
38                               minimal_key_legend=True)
39
40     #-----
41     #
42     #      Run
43     #
44     pipeline_run([second_task])
```

```

45
46
47 # modify job1.stage1
48 open("job1.a.output.1", "w").close()
49
50
51 # Print graph after everything apart from ''job1.a.output.1'' is updated
52
53 #-----
54 #
55 #       Show flow chart and tasks after running the pipeline
56 #
57 print "Show flow chart and tasks after running the pipeline"
58 pipeline_printout_graph ( open("simple_tutorial_stage5_after.png", "w"),
59                           "png",
60                           [second_task],
61                           no_key_legend=True)

```

### 1.37.2 Resulting Flowcharts



## 1.38 Chapter 8: Python Code for Specifying output file names with `formatter()` and `regex()`

See also:

- *Manual Table of Contents*
- `suffix()` syntax
- `formatter()` syntax
- `regex()` syntax
- Back to **Chapter 8: Specifying output file names**

### 1.38.1 Example Code for `suffix()`

```
from ruffus import *

#-----
#    create initial files
#
@originate([
    ['job1.a.start', 'job1.b.start'],
    ['job2.a.start', 'job2.b.start'],
    ['job3.a.start', 'job3.b.start']    ])
def create_initial_file_pairs(output_files):
    # create both files as necessary
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

#-----
#    suffix
#
@transform(create_initial_file_pairs,
           suffix(".start"),
           [".output.a.1", 45, ".output.b.1"])
def first_task(input_files, output_parameters):
    print "  input_parameters = ", input_files
    print "  output_parameters = ", output_parameters

#
#        Run
#
pipeline_run([first_task])
```

### 1.38.2 Example Code for `formatter()`

```
from ruffus import *

#    create initial files
@originate([
    ['job1.a.start', 'job1.b.start'],
    ['job2.a.start', 'job2.b.start'],
    ['job3.a.start', 'job3.c.start']    ])
def create_initial_file_pairs(output_files):
    # create both files as necessary
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

#-----
#    formatter
#
#    first task
@transform(create_initial_file_pairs,                                # Input
          formatter(".+/job(?P<JOBNUMBER>\d+).a.start",
                    ".+/job[123].b.start"),                                # Extract job number
                                                # Match only "b" files
```

```

        ["{path[0]}/jobs{JOBNUMBER[0]}.output.a.1",
         "{path[1]}/jobs{JOBNUMBER[0]}.output.b.1", 45])           # Replacement list
def first_task(input_files, output_parameters):
    print "input_parameters = ", input_files
    print "output_parameters = ", output_parameters

#
#       Run
#
pipeline_run(verbose=0)

```

### 1.38.3 Example Code for `formatter()` with replacements in `extra` arguments

```

from ruffus import *

#   create initial files
@originate([
    ['job1.a.start', 'job1.b.start'],
    ['job2.a.start', 'job2.b.start'],
    ['job3.a.start', 'job3.c.start']    ])
def create_initial_file_pairs(output_files):
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

#-----
#   print job number as an extra argument
#

#   first task
@transform(create_initial_file_pairs,                                # Input
          formatter(".+/job(?P<JOBNR>\d+).a.start",
                    ".+/job[123].b.start"),                         # Extract job number
          [{"path[0]}/jobs{JOBNUMBER[0]}.output.a.1",
           "{path[1]}/jobs{JOBNUMBER[0]}.output.b.1"],           # Match only "b" files
           {"{JOBNR}": ""}                                       # Replacement list
          )
def first_task(input_files, output_parameters, job_number):
    print job_number, ":", input_files

pipeline_run(verbose=0)

```

### 1.38.4 Example Code for `formatter()` in Zoos

```

from ruffus import *

#   Make directories
@mkdir(["tiger", "lion", "dog", "crocodile", "rose"])

@originate(
    #   List of animals and plants

```

```
[    "tiger/mammals.wild.animals",
    "lion/mammals.wild.animals",
    "lion/mammals.handreared.animals",
    "dog/mammals.tame.animals",
    "dog/mammals.wild.animals",
    "crocodile/reptiles.wild.animals",
    "rose/flowering.handreared.plants"])
def create_initial_files(output_file):
    with open(output_file, "w") as oo: pass

# Put different animals in different directories depending on their clade
@transform(create_initial_files,
           # Input
           formatter(".+/ (?P<clade>\w+). (?P<tame>\w+).animals"),
           # Only animals: ignore plants
           "{subpath[0][1]}/{clade[0]}/{tame[0]}.{subdir[0][0]}.food", # Replacement
           "{subpath[0][1]}/{clade[0]}",
           "{subdir[0][0]}",
           "{tame[0]}") # new_directory
# animal_name
# tameness
def feed(input_file, output_file, new_directory, animal_name, tameness):
    print "Food for the {tameness:11s} {animal_name:9s} = {output_file:90s} will be placed in {new_directory:90s}"
    pipeline_run(verbose=0)

Results in:
::

>>> pipeline_run(verbose=0)
Food for the wild      crocodile = ./reptiles/wild.crocodile.food will be placed in ./reptiles/wild
Food for the tame      dog       = ./mammals/tame.dog.food      will be placed in ./mammals/tame
Food for the wild      dog       = ./mammals/wild.dog.food     will be placed in ./mammals/wild
Food for the handreared lion     = ./mammals/handreared.lion.food will be placed in ./mammals/handreared
Food for the wild      lion     = ./mammals/wild.lion.food     will be placed in ./mammals/wild
Food for the wild      tiger   = ./mammals/wild.tiger.food    will be placed in ./mammals/wild
```

### 1.38.5 Example Code for `regex()` in zoos

```
from ruffus import *

# Make directories
@mkdir(["tiger", "lion", "dog", "crocodile", "rose"])

@originate(
    # List of animals and plants
    [    "tiger/mammals.wild.animals",
        "lion/mammals.wild.animals",
        "lion/mammals.handreared.animals",
        "dog/mammals.tame.animals",
        "dog/mammals.wild.animals",
        "crocodile/reptiles.wild.animals",
        "rose/flowering.handreared.plants"])
```

```

def create_initial_files(output_file):
    with open(output_file, "w") as oo: pass

# Put different animals in different directories depending on their clade
@transform(create_initial_files,                                         # Input
          regex(r"(.*/?) (\w+)/ (?P<clade>\w+) . (?P<tame>\w+).animals"), # Only animals: ignore p
          r"\1/\g<clade>/\g<tame>.\\2.food",                                # Replacement
          r"\1/\g<clade>",                                                 # new_directory
          r"\2",                                                       # animal_name
          "\g<tame>")                                              # tameness

def feed(input_file, output_file, new_directory, animal_name, tameness):
    print "Food for the {tameness:11s} {animal_name:9s} = {output_file:90s} will be placed in {n

pipeline_run(verbose=0)

Results in:
::

>>> pipeline_run(verbose=0)
Food for the wild      crocodile = reptiles/wild.crocodile.food will be placed in reptiles
Food for the tame      dog      = mammals/tame.dog.food      will be placed in mammals
Food for the wild      dog      = mammals/wild.dog.food      will be placed in mammals
Food for the handreared lion    = mammals/handreared.lion.food will be placed in mammals
Food for the wild      lion    = mammals/wild.lion.food      will be placed in mammals
Food for the wild      tiger   = mammals/wild.tiger.food      will be placed in mammals

```

## 1.39 Chapter 9: Python Code for Preparing directories for output with `@mkdir()`

#### **See also:**

- *Manual Table of Contents*
  - `mkdir()` syntax
  - `formatter()` syntax
  - `regex()` syntax
  - Back to **Chapter 9: Preparing directories for output with `@mkdir()`**

### 1.39.1 Code for `formatter()` Zoo example

```
from ruffus import *
# Make directories
@mkdir(["tiger", "lion", "dog", "crocodile", "rose"])
```

```
@originate(
    # List of animals and plants
    [
        "tiger/mammals.wild.animals",
        "lion/mammals.wild.animals",
        "lion/mammals.handreared.animals",
        "dog/mammals.tame.animals",
        "dog/mammals.wild.animals",
        "crocodile/reptiles.wild.animals",
        "rose/flowering.handreared.plants"])
def create_initial_files(output_file):
    with open(output_file, "w") as oo: pass

# create directories for each clade
@mkdir(    create_initial_files,                                     # Input
         formatter(".+/(?P<clade>\w+).(?P<tame>\w+).animals"),      # Only animals: ignore plants
         "{subpath[0][1]}/{clade[0]}")                                       # new_directory
# Put different animals in different directories depending on their clade
@transform(create_initial_files,                                     # Input
          formatter(".+/(?P<clade>\w+).(?P<tame>\w+).animals"),      # Only animals: ignore plants
          "{subpath[0][1]}/{clade[0]}/{tame[0]}.{subdir[0][0]}.food", # Replacement
          "{subpath[0][1]}/{clade[0]}",
          "{subdir[0][0]}",                                              # animal_name
          "{tame[0]}")                                                 # tameness

def feed(input_file, output_file, new_directory, animal_name, tameness):
    print "%40s -> %90s" % (input_file, output_file)
    # this works now
    open(output_file, "w")

pipeline_run(verbose=0)
```

### 1.39.2 Code for `regex()` Zoo example

```
from ruffus import *

# Make directories
@mkdir(["tiger", "lion", "dog", "crocodile", "rose"])
@originate(
    # List of animals and plants
    [
        "tiger/mammals.wild.animals",
        "lion/mammals.wild.animals",
        "lion/mammals.handreared.animals",
        "dog/mammals.tame.animals",
        "dog/mammals.wild.animals",
        "crocodile/reptiles.wild.animals",
        "rose/flowering.handreared.plants"])
def create_initial_files(output_file):
    with open(output_file, "w") as oo: pass
```

```
# create directories for each clade
@mkdir(    create_initial_files,                                     # Input

          regex(r"(.*/?) (\w+) / (?P<clade>\w+).(?P<tame>\w+).animals"), # Only animals: ignore p
          r"\g<clade>" )                                                       # new_directory

# Put different animals in different directories depending on their clade
@transform(create_initial_files,                                         # Input

          regex(r"(.*/?) (\w+) / (?P<clade>\w+).(?P<tame>\w+).animals"), # Only animals: ignore p
          r"\1\g<clade>/\g<tame>.2.food",                                       # Replacement

          r"\1\g<clade>",
          r"\2",
          "\g<tame>")

def feed(input_file, output_file, new_directory, animal_name, tameness):
    print "%40s -> %90s" % (input_file, output_file)
    # this works now
    open(output_file, "w")

pipeline_run(verbose=0)
```

## 1.40 Chapter 10: Python Code for Checkpointing: Interrupted Pipelines and Exceptions

See also:

- *Manual Table of Contents*
- Back to `\new_manual.checkpointing.chapter_num`: *Interrupted Pipelines and Exceptions*

### 1.40.1 Code for `.:ref:suffix() <decorators.suffix>` example

```
from ruffus import *
```

## 1.41 Chapter 12: Python Code for Splitting up large tasks / files with `@split`

See also:

- *Manual Table of Contents*
- `@split` syntax in detail
- Back to **Chapter 12: Splitting up large tasks / files with `@split`**

### 1.41.1 Splitting large jobs

```
from ruffus import *

NUMBER_OF_RANDOMS = 10000
CHUNK_SIZE = 1000

import random, os, glob

#-----
# Create random numbers
#
@originate("random_numbers.list")
def create_random_numbers(output_file_name):
    f = open(output_file_name, "w")
    for i in range(NUMBER_OF_RANDOMS):
        f.write("%g\n" % (random.random() * 100.0))

#-----
# split initial file
#
@splits(create_random_numbers, "*.chunks")
def split_problem (input_file_names, output_files):
    """
        splits random numbers file into xxx files of chunk_size each
    """
    #
    # clean up any files from previous runs
    #
    #for ff in glob.glob("*.chunks"):
    for ff in input_file_names:
        os.unlink(ff)
    #
    # create new file every chunk_size lines and
    #     copy each line into current file
    #
    output_file = None
    cnt_files = 0
    for input_file_name in input_file_names:
        for i, line in enumerate(open(input_file_name)):
            if i % CHUNK_SIZE == 0:
                cnt_files += 1
                output_file = open("%d.chunks" % cnt_files, "w")
                output_file.write(line)

#-----
# Calculate sum and sum of squares for each chunk file
#
@transform(split_problem, suffix(".chunks"), ".sums")
def sum_of_squares (input_file_name, output_file_name):
    output = open(output_file_name, "w")
    sum_squared, sum = [0.0, 0.0]
    cnt_values = 0
    for line in open(input_file_name):
        cnt_values += 1
```

```

        val = float(line.rstrip())
        sum_squared += val * val
        sum += val
    output.write("%s\n%s\n%d\n" % (repr(sum_squared), repr(sum), cnt_values))

#-----
#
#       Run
#
pipeline_run()

```

## 1.41.2 Resulting Output

```

>>> pipeline_run()
Job   = [None -> random_numbers.list] completed
Completed Task = create_random_numbers
Job   = [[random_numbers.list] -> *.chunks] completed
Completed Task = split_problem
Job   = [1.chunks -> 1.sums] completed
Job   = [10.chunks -> 10.sums] completed
Job   = [2.chunks -> 2.sums] completed
Job   = [3.chunks -> 3.sums] completed
Job   = [4.chunks -> 4.sums] completed
Job   = [5.chunks -> 5.sums] completed
Job   = [6.chunks -> 6.sums] completed
Job   = [7.chunks -> 7.sums] completed
Job   = [8.chunks -> 8.sums] completed
Job   = [9.chunks -> 9.sums] completed
Completed Task = sum_of_squares

```

## 1.42 Chapter 13: Python Code for @merge multiple input into a single result

See also:

- *Manual Table of Contents*
- *@merge syntax in detail*
- Back to [Chapter 13: Splitting up large tasks / files with @merge](#)

### 1.42.1 Splitting large jobs

```

from ruffus import *

NUMBER_OF_RANDOMS = 10000
CHUNK_SIZE = 1000

import random, os, glob

#-----
#

```

```
#      Create random numbers
#
@originate("random_numbers.list")
def create_random_numbers(output_file_name):
    f = open(output_file_name, "w")
    for i in range(NUMBER_OF_RANDOMS):
        f.write("%g\n" % (random.random() * 100.0))

#-----
#
#      split initial file
#
@split(create_random_numbers, "*.chunks")
def split_problem (input_file_names, output_files):
    """
        splits random numbers file into xxx files of chunk_size each
    """
    #
    #      clean up any files from previous runs
    #
    #for ff in glob.glob("*.chunks"):
    for ff in input_file_names:
        os.unlink(ff)
    #
    #
    #      create new file every chunk_size lines and
    #          copy each line into current file
    #
    output_file = None
    cnt_files = 0
    for input_file_name in input_file_names:
        for i, line in enumerate(open(input_file_name)):
            if i % CHUNK_SIZE == 0:
                cnt_files += 1
                output_file = open("%d.chunks" % cnt_files, "w")
                output_file.write(line)

#-----
#
#      Calculate sum and sum of squares for each chunk file
#
@transform(split_problem, suffix(".chunks"), ".sums")
def sum_of_squares (input_file_name, output_file_name):
    output = open(output_file_name, "w")
    sum_squared, sum = [0.0, 0.0]
    cnt_values = 0
    for line in open(input_file_name):
        cnt_values += 1
        val = float(line.rstrip())
        sum_squared += val * val
        sum += val
    output.write("%s\n%s\n%d\n" % (repr(sum_squared), repr(sum), cnt_values))

#-----
#
#      Calculate variance from sums
#
@merge(sum_of_squares, "variance.result")
```

```

def calculate_variance (input_file_names, output_file_name):
    """
    Calculate variance naively
    """
    #
    # initialise variables
    #
    all_sum_squared = 0.0
    all_sum = 0.0
    all_cnt_values = 0.0
    #
    # added up all the sum_squared, and sum and cnt_values from all the chunks
    #
    for input_file_name in input_file_names:
        sum_squared, sum, cnt_values = map(float, open(input_file_name).readlines())
        all_sum_squared += sum_squared
        all_sum += sum
        all_cnt_values += cnt_values
    all_mean = all_sum / all_cnt_values
    variance = (all_sum_squared - all_sum * all_mean)/(all_cnt_values)
    #
    # print output
    #
    open(output_file_name, "w").write("%s\n" % variance)

#-----
#
#       Run
#
pipeline_run()

```

## 1.42.2 Resulting Output

```

>>> pipeline_run()
Job = [None -> random_numbers.list] completed
Completed Task = create_random_numbers
Job = [[random_numbers.list] -> *.chunks] completed
Completed Task = split_problem
Job = [1.chunks -> 1.sums] completed
Job = [10.chunks -> 10.sums] completed
Job = [2.chunks -> 2.sums] completed
Job = [3.chunks -> 3.sums] completed
Job = [4.chunks -> 4.sums] completed
Job = [5.chunks -> 5.sums] completed
Job = [6.chunks -> 6.sums] completed
Job = [7.chunks -> 7.sums] completed
Job = [8.chunks -> 8.sums] completed
Job = [9.chunks -> 9.sums] completed
Completed Task = sum_of_squares
Job = [[1.sums, 10.sums, 2.sums, 3.sums, 4.sums, 5.sums, 6.sums, 7.sums, 8.sums, 9.sums] ->
Completed Task = calculate_variance

```

## 1.43 Chapter 14: Python Code for Multiprocessing, drmaa and Computation Clusters

See also:

- *Manual Table of Contents*
- *@jobs\_limit* syntax
- *pipeline\_run()* syntax
- *drmaa\_wrapper.run\_job()* syntax
- Back to **Chapter 14: Multiprocessing, drmaa and Computation Clusters**

### 1.43.1 @jobs\_limit

- First 2 tasks are constrained to a parallelism of 3 shared jobs at a time
- Final task is constrained to a parallelism of 5 jobs at a time
- The entire pipeline is constrained to a (theoretical) parallelism of 10 jobs at a time

```
from ruffus import *
import time

# make list of 10 files
@splits(None, "*stage1")
def make_files(input_files, output_files):
    for i in range(10):
        if i < 5:
            open("%d.small_stage1" % i, "w")
        else:
            open("%d.big_stage1" % i, "w")

@jobs_limit(3, "ftp_download_limit")
@transform(make_files, suffix(".small_stage1"), ".stage2")
def stage1_small(input_file, output_file):
    print "FTP downloading %s ->Start" % input_file
    time.sleep(2)
    open(output_file, "w")
    print "FTP downloading %s ->Finished" % input_file

@jobs_limit(3, "ftp_download_limit")
@transform(make_files, suffix(".big_stage1"), ".stage2")
def stage1_big(input_file, output_file):
    print "FTP downloading %s ->Start" % input_file
    time.sleep(2)
    open(output_file, "w")
    print "FTP downloading %s ->Finished" % input_file

@jobs_limit(5)
@transform([stage1_small, stage1_big], suffix(".stage2"), ".stage3")
def stage2(input_file, output_file):
    print "Processing stage2 %s ->Start" % input_file
    time.sleep(2)
    open(output_file, "w")
    print "Processing stage2 %s ->Finished" % input_file
```

```
pipeline_run(multiprocess = 10, verbose = 0)
```

Giving:

```
>>> pipeline_run(multiprocess = 10, verbose = 0)
```

```
>>> # 3 jobs at a time, interleaved
FTP downloading 5.big_stage1 ->Start
FTP downloading 6.big_stage1 ->Start
FTP downloading 7.big_stage1 ->Start
FTP downloading 5.big_stage1 ->Finished
FTP downloading 8.big_stage1 ->Start
FTP downloading 6.big_stage1 ->Finished
FTP downloading 9.big_stage1 ->Start
FTP downloading 7.big_stage1 ->Finished
FTP downloading 0.small_stage1 ->Start
FTP downloading 8.big_stage1 ->Finished
FTP downloading 1.small_stage1 ->Start
FTP downloading 9.big_stage1 ->Finished
FTP downloading 2.small_stage1 ->Start
FTP downloading 0.small_stage1 ->Finished
FTP downloading 3.small_stage1 ->Start
FTP downloading 1.small_stage1 ->Finished
FTP downloading 4.small_stage1 ->Start
FTP downloading 2.small_stage1 ->Finished
FTP downloading 3.small_stage1 ->Finished
FTP downloading 4.small_stage1 ->Finished
```

```
>>> # 5 jobs at a time, interleaved
Processing stage2 0.stage2 ->Start
Processing stage2 1.stage2 ->Start
Processing stage2 2.stage2 ->Start
Processing stage2 3.stage2 ->Start
Processing stage2 4.stage2 ->Start
Processing stage2 0.stage2 ->Finished
Processing stage2 5.stage2 ->Start
Processing stage2 1.stage2 ->Finished
Processing stage2 6.stage2 ->Start
Processing stage2 2.stage2 ->Finished
Processing stage2 4.stage2 ->Finished
Processing stage2 7.stage2 ->Start
Processing stage2 8.stage2 ->Start
Processing stage2 3.stage2 ->Finished
Processing stage2 9.stage2 ->Start
Processing stage2 5.stage2 ->Finished
Processing stage2 7.stage2 ->Finished
Processing stage2 6.stage2 ->Finished
Processing stage2 8.stage2 ->Finished
Processing stage2 9.stage2 ->Finished
```

### 1.43.2 Using ruffus.drmaa\_wrapper

```
#!/usr/bin/python
job_queue_name      = "YOUR_QUEUE_NAME_GOES_HERE"
job_other_options  = "-P YOUR_PROJECT_NAME_GOES_HERE"
```

```
from ruffus import *
from ruffus.drmaa_wrapper import run_job, error_drmaa_job

parser = cmdline.get_argparse(description='WHAT DOES THIS PIPELINE DO?')

options = parser.parse_args()

# logger which can be passed to multiprocessing ruffus tasks
logger, logger_mutex = cmdline.setup_logging (__name__, options.log_file, options.verbose)

#
# start shared drmaa session for all jobs / tasks in pipeline
#
import drmaa
drmaa_session = drmaa.Session()
drmaa_session.initialize()

@originate(["1.chromosome", "X.chromosome"],
           logger, logger_mutex)
def create_test_files(output_file):
    try:
        stdout_res, stderr_res = "", ""
        job_queue_name, job_other_options = get_queue_options()

        #
        # ruffus.drmaa_wrapper.run_job
        #
        stdout_res, stderr_res = run_job(cmd_str = "touch " + output_file,
                                         job_name = job_name,
                                         logger = logger,
                                         drmaa_session = drmaa_session,
                                         run_locally = options.local_run,
                                         job_queue_name = job_queue_name,
                                         job_other_options = job_other_options)

        # relay all the stdout, stderr, drmaa output to diagnose failures
    except error_drmaa_job as err:
        raise Exception("\n".join(map(str,
                                      "Failed to run:",
                                      cmd,
                                      err,
                                      stdout_res,
                                      stderr_res)))

    if __name__ == '__main__':
        cmdline.run (options, multithread = options.jobs)
        # cleanup drmaa
        drmaa_session.exit()
```

## 1.44 Chapter 15: Python Code for Logging progress through a pipeline

See also:

- *Manual Table of Contents*
- Back to **Chapter 15**: *Logging progress through a pipeline*

### 1.44.1 Rotating set of file logs

```
import logging
import logging.handlers

LOG_FILENAME = '/tmp/ruffus.log'

# Set up a specific logger with our desired output level
logger = logging.getLogger('My_Ruffus_logger')
logger.setLevel(logging.DEBUG)

# Rotate a set of 5 log files every 2kb
handler = logging.handlers.RotatingFileHandler(
    LOG_FILENAME, maxBytes=2000, backupCount=5)

# Add the log message handler to the logger
logger.addHandler(handler)

# Ruffus pipeline
from ruffus import *

# Start with some initial data file of yours...
initial_file = "job1.input"
open(initial_file, "w")

@transform( initial_file,
           suffix(".input"),
           ".output1"),
def first_task(input_file, output_file):
    "Some detailed description"
    pass

# use our custom logging object
pipeline_run(logger=logger)
print open("/tmp/ruffus.log").read()
```

## 1.45 Chapter 16: Python Code for @subdivide tasks to run efficiently and regroup with @collate

See also:

- *Manual Table of Contents*
- *@jobs\_limit* syntax
- *pipeline\_run()* syntax
- *drmaa\_wrapper.run\_job()* syntax
- Back to **Chapter 16**: :ref:`@subdivide tasks to run efficiently and regroup with @collate`

### 1.45.1 @subdivide and regroup with @collate example

```
from ruffus import *
import os, random, sys

# Create files a random number of lines
@originate(["a.start",
            "b.start",
            "c.start"])
def create_test_files(output_file):
    cnt_lines = random.randint(1,3) * 2
    with open(output_file, "w") as oo:
        for ii in range(cnt_lines):
            oo.write("data item = %d\n" % ii)
    print "%s has %d lines" % (output_file, cnt_lines)

#
#    subdivide the input files into NNN fragment files of 2 lines each
#
@subdivide( create_test_files,
            formatter(),
            "{path[0]}/{basename[0]}.*.fragment",
            "{path[0]}/{basename[0]}")
def subdivide_files(input_file, output_files, output_file_name_stem):
    #
    # cleanup any previous results
    #
    for oo in output_files:
        os.unlink(oo)
    #
    # Output files contain two lines each
    #      (new output files every even line)
    #
    cnt_output_files = 0
    for ii, line in enumerate(open(input_file)):
        if ii % 2 == 0:
            cnt_output_files += 1
            output_file_name = "%s.%d.fragment" % (output_file_name_stem, cnt_output_files)
            output_file = open(output_file_name, "w")
            print "Subdivide %s -> %s" % (input_file, output_file_name)
            output_file.write(line)

    #
    # Analyse each fragment independently
    #
@transform(subdivide_files, suffix(".fragment"), ".analysed")
def analyse_fragments(input_file, output_file):
    print "Analysing %s -> %s" % (input_file, output_file)
    with open(output_file, "w") as oo:
        for line in open(input_file):
            oo.write("analysed " + line)

    #
    # Group results using original names
    #
```

```

@collate(    analyse_fragments,
             # split file name into [abc].NUMBER.analysed
             formatter("/(?:P<NAME>[abc]+)\.\d+\.analysed$"),
             "{path[0]}/{NAME[0]}.final_result")
def recombine_analyses(input_file_names, output_file):
    with open(output_file, "w") as oo:
        for input_file in input_file_names:
            print "      Recombine %s -> %s" % (input_file, output_file)
            for line in open(input_file):
                oo.write(line)

#pipeline_printout(sys.stdout, verbose = 3)

pipeline_run(verbose = 1)

```

Results in

```

>>> pipeline_run(verbose = 1)

    a.start has 2 lines
Job   = [None -> a.start] completed
    b.start has 6 lines
Job   = [None -> b.start] completed
    c.start has 6 lines
Job   = [None -> c.start] completed
Completed Task = create_test_files

    Subdivide a.start -> /home/lg/temp/a.1.fragment
Job   = [a.start -> a.*.fragment, a] completed
    Subdivide b.start -> /home/lg/temp/b.1.fragment
    Subdivide b.start -> /home/lg/temp/b.2.fragment
    Subdivide b.start -> /home/lg/temp/b.3.fragment
Job   = [b.start -> b.*.fragment, b] completed
    Subdivide c.start -> /home/lg/temp/c.1.fragment
    Subdivide c.start -> /home/lg/temp/c.2.fragment
    Subdivide c.start -> /home/lg/temp/c.3.fragment
Job   = [c.start -> c.*.fragment, c] completed
Completed Task = subdivide_files

    Analysing /home/lg/temp/a.1.fragment -> /home/lg/temp/a.1.analysed
Job   = [a.1.fragment -> a.1.analysed] completed
    Analysing /home/lg/temp/b.1.fragment -> /home/lg/temp/b.1.analysed
Job   = [b.1.fragment -> b.1.analysed] completed
    Analysing /home/lg/temp/b.2.fragment -> /home/lg/temp/b.2.analysed
Job   = [b.2.fragment -> b.2.analysed] completed
    Analysing /home/lg/temp/b.3.fragment -> /home/lg/temp/b.3.analysed
Job   = [b.3.fragment -> b.3.analysed] completed
    Analysing /home/lg/temp/c.1.fragment -> /home/lg/temp/c.1.analysed
Job   = [c.1.fragment -> c.1.analysed] completed
    Analysing /home/lg/temp/c.2.fragment -> /home/lg/temp/c.2.analysed
Job   = [c.2.fragment -> c.2.analysed] completed
    Analysing /home/lg/temp/c.3.fragment -> /home/lg/temp/c.3.analysed

```

```
Job = [c.3.fragment -> c.3.analysed] completed
Completed Task = analyse_fragments

Recombine /home/lg/temp/a.1.analysed -> /home/lg/temp/a.final_result
Job = [[a.1.analysed] -> a.final_result] completed
Recombine /home/lg/temp/b.1.analysed -> /home/lg/temp/b.final_result
Recombine /home/lg/temp/b.2.analysed -> /home/lg/temp/b.final_result
Recombine /home/lg/temp/b.3.analysed -> /home/lg/temp/b.final_result
Job = [[b.1.analysed, b.2.analysed, b.3.analysed] -> b.final_result] completed
Recombine /home/lg/temp/c.1.analysed -> /home/lg/temp/c.final_result
Recombine /home/lg/temp/c.2.analysed -> /home/lg/temp/c.final_result
Recombine /home/lg/temp/c.3.analysed -> /home/lg/temp/c.final_result
Job = [[c.1.analysed, c.2.analysed, c.3.analysed] -> c.final_result] completed
Completed Task = recombine_analyses
```

## 1.46 Chapter 17: Python Code for @combinations, @permutations and all versus all @product

See also:

- *Manual Table of Contents*
- *@combinations\_with\_replacement*
- *@combinations*
- *@permutations*
- *@product*
- Back to **Chapter 17: Preparing directories for output with @combinatorics()**

### 1.46.1 Example code for @product

```
from ruffus import *
from ruffus.combinatorics import *

# Three sets of initial files
@originate(['a.start', 'b.start'])
def create_initial_files_ab(output_file):
    with open(output_file, "w") as oo: pass

@originate(['p.start', 'q.start'])
def create_initial_files_pq(output_file):
    with open(output_file, "w") as oo: pass

@originate([('x.1_start', 'x.2_start'),
            ['y.1_start', 'y.2_start'] ])
def create_initial_files_xy(output_file):
    with open(output_file, "w") as oo: pass

# @product
@product(    create_initial_files_ab,           # Input
           formatter("(..start)$"),          # match input file set # 1
           create_initial_files_pq,          # Input
```

```

formatter("(..start)$"),           # match input file set # 2
create_initial_files_xy,          # Input
formatter("(..start)$"),          # match input file set # 3

" {path[0][0] }/"                # Output Replacement string
" {basename[0][0] }_vs_"          #
" {basename[1][0] }_vs_"          #
" {basename[2][0] }.product",     #

" {path[0][0] }",                 # Extra parameter: path for 1st set of files, 1st fi
["{basename[0][0] }",             # Extra parameter: basename for 1st set of files, 1s
 " {basename[1][0] }",             # 2nd
 " {basename[2][0] }",             # 3rd
])

def product_task(input_file, output_parameter, shared_path, basenames):
    print "# basenames      = ", ".join(basenames)
    print "input_parameter = ", input_file
    print "output_parameter = ", output_parameter, "\n"

#
#       Run
#
pipeline_run(verbose=0)

```

This results in:

```

>>> pipeline_run(verbose=0)

# basenames      = a p x
input_parameter  = ('a.start', 'p.start', 'x.start')
output_parameter = /home/lg/temp/a_vs_p_vs_x.product

# basenames      = a p y
input_parameter  = ('a.start', 'p.start', 'y.start')
output_parameter = /home/lg/temp/a_vs_p_vs_y.product

# basenames      = a q x
input_parameter  = ('a.start', 'q.start', 'x.start')
output_parameter = /home/lg/temp/a_vs_q_vs_x.product

# basenames      = a q y
input_parameter  = ('a.start', 'q.start', 'y.start')
output_parameter = /home/lg/temp/a_vs_q_vs_y.product

# basenames      = b p x
input_parameter  = ('b.start', 'p.start', 'x.start')
output_parameter = /home/lg/temp/b_vs_p_vs_x.product

# basenames      = b p y
input_parameter  = ('b.start', 'p.start', 'y.start')
output_parameter = /home/lg/temp/b_vs_p_vs_y.product

# basenames      = b q x
input_parameter  = ('b.start', 'q.start', 'x.start')
output_parameter = /home/lg/temp/b_vs_q_vs_x.product

```

```
# basenames      = b q y
input_parameter = ('b.start', 'q.start', 'y.start')
output_parameter = /home/lg/temp/b_vs_q_vs_y.product
```

## 1.46.2 Example code for `@permutations`

```
from ruffus import *
from ruffus.combinatorics import *

# initial file pairs
@originate([
    ['A.1_start', 'A.2_start'],
    ['B.1_start', 'B.2_start'],
    ['C.1_start', 'C.2_start'],
    ['D.1_start', 'D.2_start']])
def create_initial_files_ABCD(output_files):
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

# @permutations
@permutations(create_initial_files_ABCD,
              formatter(),
              # Input
              # match input files
              # tuple of 2 at a time
              2,
              # Output Replacement string
              "{path[0][0]}/"
              "{basename[0][1]}_vs_"
              "{basename[1][1]}.permutations",
              # Extra parameter: path for 1st set of files, 1st file name
              "{path[0][0]}",
              # Extra parameter
              ["{basename[0][0]}", # basename for 1st set of files, 1st file name
               "{basename[1][0]}", # 2nd
               ])
def permutations_task(input_file, output_parameter, shared_path, basenames):
    print " - ".join(basenames)

#
#       Run
#
pipeline_run(verbose=0)
```

This results in:

```
>>> pipeline_run(verbose=0)

A - B
A - C
A - D
B - A
B - C
B - D
```

```
C - A
C - B
C - D
D - A
D - B
D - C
```

### 1.46.3 Example code for `@combinations`

```
from ruffus import *
from ruffus.combinatorics import *

# initial file pairs
@originate([
    ['A.1_start', 'A.2_start'],
    ['B.1_start', 'B.2_start'],
    ['C.1_start', 'C.2_start'],
    ['D.1_start', 'D.2_start']])
def create_initial_files_ABCD(output_files):
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

# @combinations
@combinations(create_initial_files_ABCD,           # Input
              formatter(),                      # match input files

              # tuple of 3 at a time
              3,

              # Output Replacement string
              "{path[0][0]}/"
              "{basename[0][1]}_vs_"
              "{basename[1][1]}_vs_"
              "{basename[2][1]}.combinations",

              # Extra parameter: path for 1st set of files, 1st file name
              "{path[0][0]}",

              # Extra parameter
              ["{basename[0][0]}",   # basename for 1st set of files, 1st file name
               "{basename[1][0]}",   # 2nd
               "{basename[2][0]}",   # 3rd
               ])

def combinations_task(input_file, output_parameter, shared_path, basenames):
    print " - ".join(basenames)

#
#       Run
#
pipeline_run(verbose=0)
```

This results in:

```
>>> pipeline_run(verbose=0)
A - B - C
A - B - D
A - C - D
```

B - C - D

#### 1.46.4 Example code for `@combinations_with_replacement`

```
from ruffus import *
from ruffus.combinatorics import *

# initial file pairs
@originate([
    ['A.1_start', 'A.2_start'],
    ['B.1_start', 'B.2_start'],
    ['C.1_start', 'C.2_start'],
    ['D.1_start', 'D.2_start']])
def create_initial_files_ABCD(output_files):
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

# @combinations_with_replacement
@combinations_with_replacement(create_initial_files_ABCD,           # Input
                                formatter(),                      # match input files

                                # tuple of 2 at a time
                                2,

                                # Output Replacement string
                                "{path[0][0]}/"
                                "{basename[0][1]}_vs_"
                                "{basename[1][1]}.combinations_with_replacement",

                                # Extra parameter: path for 1st set of files, 1st file name
                                "{path[0][0]}",

                                # Extra parameter
                                ["{basename[0][0]}",  # basename for 1st set of files, 1st file name
                                 "{basename[1][0]}",  #                         2rd
                                 ]]

def combinations_with_replacement_task(input_file, output_parameter, shared_path, basenames):
    print " - ".join(basenames)

#
#      Run
#
pipeline_run(verbose=0)
```

This results in:

```
>>> pipeline_run(verbose=0)
A - A
A - B
A - C
A - D
B - B
B - C
B - D
C - C
C - D
D - D
```

## 1.47 Chapter 20: Python Code for Manipulating task inputs via string substitution using *inputs()* and *add\_inputs()*

See also:

- *Manual Table of Contents*
- *inputs()* syntax
- *add\_inputs()* syntax
- Back to **Chapter 20: Manipulating task inputs via string substitution**

### 1.47.1 Example code for adding additional *input* prerequisites per job with *add\_inputs()*

#### 1. Example: compiling c++ code

```
# source files exist before our pipeline
source_files = ["hasty.cpp", "tasty.cpp", "messy.cpp"]
for source_file in source_files:
    open(source_file, "w")

from ruffus import *

@transform(source_files, suffix(".cpp"), ".o")
def compile(input_filename, output_file):
    open(output_file, "w")

pipeline_run()
```

Giving:

```
>>> pipeline_run()
Job  = [hasty.cpp -> hasty.o] completed
Job  = [messy.cpp -> messy.o] completed
Job  = [tasty.cpp -> tasty.o] completed
Completed Task = compile
```

#### 2. Example: Adding a common header file with *add\_inputs()*

```
# source files exist before our pipeline
source_files = ["hasty.cpp", "tasty.cpp", "messy.cpp"]
for source_file in source_files:
    open(source_file, "w")

# common (universal) header exists before our pipeline
open("universal.h", "w")

from ruffus import *

@transform( source_files, suffix(".cpp"),
           # add header to the input of every job
           add_inputs("universal.h"),
           ".o")
```

```
def compile(input_filename, output_file):
    open(output_file, "w")

pipeline_run()
```

Giving:

```
>>> pipeline_run()
Job   = [[hasty.cpp, universal.h] -> hasty.o] completed
Job   = [[messy.cpp, universal.h] -> messy.o] completed
Job   = [[tasty.cpp, universal.h] -> tasty.o] completed
Completed Task = compile
```

### 3. Example: Additional *Input* can be tasks

```
# source files exist before our pipeline
source_files = ["hasty.cpp", "tasty.cpp", "messy.cpp"]
for source_file in source_files:
    open(source_file, "w")

# common (universal) header exists before our pipeline
open("universal.h", "w")

from ruffus import *

# make header files
@transform(source_files, suffix(".cpp"), ".h")
def create_matching_headers(input_file, output_file):
    open(output_file, "w")

@transform(source_files, suffix(".cpp"),
           # add header to the input of every job
           add_inputs("universal.h",
                      # add result of task create_matching_headers to the input of every job
                      create_matching_headers,
                      ".o"))
def compile(input_filename, output_file):
    open(output_file, "w")

pipeline_run()
```

Giving:

```
>>> pipeline_run()
Job   = [hasty.cpp -> hasty.h] completed
Job   = [messy.cpp -> messy.h] completed
Job   = [tasty.cpp -> tasty.h] completed
Completed Task = create_matching_headers
Job   = [[hasty.cpp, universal.h, hasty.h, messy.h, tasty.h] -> hasty.o] completed
Job   = [[messy.cpp, universal.h, hasty.h, messy.h, tasty.h] -> messy.o] completed
Job   = [[tasty.cpp, universal.h, hasty.h, messy.h, tasty.h] -> tasty.o] completed
Completed Task = compile
```

#### 4. Example: Add corresponding files using `add_inputs()` with `formatter` or `regex`

```
# source files exist before our pipeline
source_files = ["hasty.cpp", "tasty.cpp", "messy.cpp"]
header_files = ["hasty.h", "tasty.h", "messy.h"]
for source_file in source_files + header_files:
    open(source_file, "w")

# common (universal) header exists before our pipeline
open("universal.h", "w")

from ruffus import *

@transform( source_files,
            formatter(".cpp$"),
            # corresponding header for each source file
            add_inputs("{basename[0]}.h",
                      # add header to the input of every job
                      "universal.h"),
            "{basename[0]}.o")
def compile(input_filename, output_file):
    open(output_file, "w")

pipeline_run()
```

Giving:

```
>>> pipeline_run()
Job  = [[hasty.cpp, hasty.h, universal.h] -> hasty.o] completed
Job  = [[messy.cpp, messy.h, universal.h] -> messy.o] completed
Job  = [[tasty.cpp, tasty.h, universal.h] -> tasty.o] completed
Completed Task = compile
```

#### 1.47.2 Example code for replacing all input parameters with `inputs()`

##### 5. Example: Running matching python scripts using `inputs()`

```
# source files exist before our pipeline
source_files = ["hasty.cpp", "tasty.cpp", "messy.cpp"]
python_files = ["hasty.py", "tasty.py", "messy.py"]
for source_file in source_files + python_files:
    open(source_file, "w")

# common (universal) header exists before our pipeline
open("universal.h", "w")

from ruffus import *

@transform( source_files,
            formatter(".cpp$"),
            # corresponding python file for each source file
            inputs("{basename[0]}.py"),
            "{basename[0]}.results")
def run_corresponding_python(input_filenames, output_file):
    open(output_file, "w")
```

```
pipeline_run()
```

Giving:

```
>>> pipeline_run()
Job   = [hasty.py -> hasty.results] completed
Job   = [messy.py -> messy.results] completed
Job   = [tasty.py -> tasty.results] completed
Completed Task = run_corresponding_python
```

## 1.48 Chapter 21: Esoteric: Python Code for Generating parameters on the fly with @files

See also:

- *Manual Table of Contents*
- *@files on-the-fly syntax in detail*
- Back to **Chapter 21: Generating parameters on the fly**

### 1.48.1 Introduction

This script takes N pairs of input file pairs (with the suffixes .gene and .gwas) and runs them against M sets of simulation data (with the suffix .simulation)  
A summary per input file pair is then produced

In pseudo-code:

```
STEP_1:
for n_file in NNN_pairs_of_input_files:
    for m_file in MMM_simulation_data:
        [n_file.gene,
         n_file.gwas,
         m_file.simulation] -> n_file.m_file.simulation_res

STEP_2:
for n_file in NNN_pairs_of_input_files:
    n_file.*.simulation_res -> n_file.mean

n = CNT_GENE_GWAS_FILES
m = CNT_SIMULATION_FILES
```

### 1.48.2 Code

```

from ruffus import *
import os

# constants

working_dir = "temp_NxM"
simulation_data_dir = os.path.join(working_dir, "simulation")
gene_data_dir = os.path.join(working_dir, "gene")
CNT_GENE_GWAS_FILES = 2
CNT_SIMULATION_FILES = 3

# imports

import os, sys
from itertools import izip
import glob
# Functions

# get gene gwas file pairs
#
#
#
def get_gene_gwas_file_pairs( ):
    """
    Helper function to get all *.gene, *.gwas from the direction specified
    in --gene_data_dir

    Returns
        file pairs with both .gene and .gwas extensions,
        corresponding roots (no extension) of each file
    """
    gene_files = glob.glob(os.path.join(gene_data_dir, "*.gene"))
    gwas_files = glob.glob(os.path.join(gene_data_dir, "*.gwas"))
    #
    common_roots = set(map(lambda x: os.path.splitext(os.path.split(x)[1])[0], gene_files))
    common_roots &= set(map(lambda x: os.path.splitext(os.path.split(x)[1])[0], gwas_files))
    common_roots = list(common_roots)
    #
    p = os.path; g_dir = gene_data_dir
    file_pairs = [[p.join(g_dir, x + ".gene"), p.join(g_dir, x + ".gwas")] for x in common_roots]
    return file_pairs, common_roots
#
#

```

```
# get simulation files
#
#-----#
def get_simulation_files( ):
    """
    Helper function to get all *.simulation from the direction specified
    in --simulation_data_dir

    Returns
        file with .simulation extensions,
        corresponding roots (no extension) of each file
    """
    simulation_files = glob.glob(os.path.join(simulation_data_dir, "*simulation"))
    simulation_roots = map(lambda x: os.path.splitext(os.path.split(x)[1])[0], simulation_files)
    return simulation_files, simulation_roots

#-----#
# Main logic

#-----#
# setup_simulation_data
#
#-----#
# mkdir: makes sure output directories exist before task
#
@follows(mkdir(gene_data_dir, simulation_data_dir))
def setup_simulation_data () :
    """
    create simulation files
    """
    for i in range(CNT_GENE_GWAS_FILES):
        open(os.path.join(gene_data_dir, "%03d.gene" % i), "w")
        open(os.path.join(gene_data_dir, "%03d.gwas" % i), "w")
    #
    # gene files without corresponding gwas and vice versa
    open(os.path.join(gene_data_dir, "orphan1.gene"), "w")
    open(os.path.join(gene_data_dir, "orphan2.gwas"), "w")
    open(os.path.join(gene_data_dir, "orphan3.gwas"), "w")
    #
    for i in range(CNT_SIMULATION_FILES):
        open(os.path.join(simulation_data_dir, "%03d.simulation" % i), "w")
```

---

```

#_____
#
#   cleanup_simulation_data
#
#_____
def try_rmdir (d):
    if os.path.exists(d):
        try:
            os.rmdir(d)
        except OSError:
            sys.stderr.write("Warning:\t%s is not empty and will not be removed.\n" % d)

def cleanup_simulation_data () :
    """
    cleanup files
    """
    sys.stderr.write("Cleanup working directory and simulation files.\n")
    #
    #   cleanup gene and gwas files
    #
    for f in glob.glob(os.path.join(gene_data_dir, "*.gene")):
        os.unlink(f)
    for f in glob.glob(os.path.join(gene_data_dir, "*.gwas")):
        os.unlink(f)
    try_rmdir(gene_data_dir)
    #
    #   cleanup simulation
    #
    for f in glob.glob(os.path.join(simulation_data_dir, "*simulation")):
        os.unlink(f)
    try_rmdir(simulation_data_dir)
    #
    #   cleanup working_dir
    #
    for f in glob.glob(os.path.join(working_dir, "simulation_results", "*.simulation_res")):
        os.unlink(f)
    try_rmdir(os.path.join(working_dir, "simulation_results"))
    #
    for f in glob.glob(os.path.join(working_dir, "*.mean")):
        os.unlink(f)
    try_rmdir(working_dir)

#
#
# Step 1:
#
#       for n_file in NNN_pairs_of_input_files:
#           for m_file in MMM_simulation_data:
#
#               [n_file.gene,
#                n_file.gwas,
#                m_file.simulation] -> working_dir/n_file.m_file.simulation_res
#
#
def generate_simulation_params ():


```

---

```
"""
Custom function to generate
file names for gene/gwas simulation study
"""

simulation_files, simulation_file_roots      = get_simulation_files()
gene_gwas_file_pairs, gene_gwas_file_roots = get_gene_gwas_file_pairs()
#
for sim_file, sim_file_root in izip(simulation_files, simulation_file_roots):
    for (gene, gwas), gene_file_root in izip(gene_gwas_file_pairs, gene_gwas_file_roots):
        #
        result_file = "%s.%s.simulation_res" % (gene_file_root, sim_file_root)
        result_file_path = os.path.join(working_dir, "simulation_results", result_file)
        #
        yield [gene, gwas, sim_file], result_file_path, gene_file_root, sim_file_root, result_file


#
# mkdir: makes sure output directories exist before task
#
@follows(mkdir(working_dir, os.path.join(working_dir, "simulation_results")))
@files(generate_simulation_params)
def gwas_simulation(input_files, result_file_path, gene_file_root, sim_file_root, result_file):
    """
    Dummy calculation of gene gwas vs simulation data
    Normally runs in parallel on a computational cluster
    """
    (gene_file,
     gwas_file,
     simulation_data_file) = input_files
    #
    simulation_res_file = open(result_file_path, "w")
    simulation_res_file.write("%s + %s -> %s\n" % (gene_file_root, sim_file_root, result_file))

#_____
# Step 2:
#
#     Statistical summary per gene/gwas file pair
#
#     for n_file in NNN_pairs_of_input_files:
#         working_dir/simulation_results/n.*.simulation_res
#             -> working_dir/n.mean
#
#_____
@collate(gwas_simulation, regex(r"simulation_results/(\d+)\.\d+.simulation_res"), r"\1.mean")
@posttask(lambda : sys.stdout.write("\nOK\n"))
def statistical_summary (result_files, summary_file):
    """
    Simulate statistical summary
    """
    summary_file = open(summary_file, "w")
    for f in result_files:
        summary_file.write(open(f).read())
```

```

pipeline_run([setup_simulation_data], multiprocess = 5, verbose = 2)
pipeline_run([statistical_summary], multiprocess = 5, verbose = 2)

# uncomment to printout flowchar
#
# pipeline_printout(sys.stdout, [statistical_summary], verbose=2)
# graph_printout ("flowchart.jpg", "jpg", [statistical_summary])
#

cleanup_simulation_data ()

```

### 1.48.3 Resulting Output

```

>>> pipeline_run([setup_simulation_data], multiprocess = 5, verbose = 2)
    Make directories [temp_NxM/gene, temp_NxM/simulation] completed
Completed Task = setup_simulation_data_mkdir_1
    Job completed
Completed Task = setup_simulation_data

>>> pipeline_run([statistical_summary], multiprocess = 5, verbose = 2)
    Make directories [temp_NxM, temp_NxM/simulation_results] completed
Completed Task = gwas_simulation_mkdir_1
    Job = [[temp_NxM/gene/001.gene, temp_NxM/gene/001.gwas, temp_NxM/simulation/000.simulation]
    Job = [[temp_NxM/gene/000.gene, temp_NxM/gene/000.gwas, temp_NxM/simulation/000.simulation]
    Job = [[temp_NxM/gene/001.gene, temp_NxM/gene/001.gwas, temp_NxM/simulation/001.simulation]
    Job = [[temp_NxM/gene/000.gene, temp_NxM/gene/000.gwas, temp_NxM/simulation/001.simulation]
    Job = [[temp_NxM/gene/000.gene, temp_NxM/gene/000.gwas, temp_NxM/simulation/002.simulation]
    Job = [[temp_NxM/gene/001.gene, temp_NxM/gene/001.gwas, temp_NxM/simulation/002.simulation]
Completed Task = gwas_simulation
    Job = [[temp_NxM/simulation_results/000.000.simulation_res, temp_NxM/simulation_results/000.
    Job = [[temp_NxM/simulation_results/001.000.simulation_res, temp_NxM/simulation_results/001.

```

## 1.49 Appendix 1: Python code for Flow Chart Colours with *pipeline\_printout\_graph(...)*

See also:

- *Manual Table of Contents*
- *pipeline\_printout\_graph(...)*
- Download code
- Back to *Flowchart colours*

This example shows how flowchart colours can be customised.

### 1.49.1 Code



```
parser.add_option("--flowchart", dest="flowchart",
                  metavar="FILE",
                  type="string",
                  help="Don't actually run any commands; just print the pipeline "
                       "as a flowchart.")
parser.add_option("--colour_scheme_index", dest="colour_scheme_index",
                  metavar="INTEGER",
                  type="int",
                  help="Index of colour scheme for flow chart.")
parser.add_option("--key_legend_in_graph", dest="key_legend_in_graph",
                  action="store_true", default=False,
                  help="Print out legend and key for dependency graph.")

(options, remaining_args) = parser.parse_args()
if not options.flowchart:
    raise Exception("Missing mandatory parameter: --flowchart.\n")

# imports

from ruffus import *
from ruffus.ruffus_exceptions import JobSignalledBreak

# Pipeline

# up to date tasks
#
@check_if_upToDate (lambda : (False, ""))
def Up_to_date_task1(infile, outfile):
    pass

@check_if_upToDate (lambda : (False, ""))
@follows(Up_to_date_task1)
def Up_to_date_task2(infile, outfile):
    pass

@check_if_upToDate (lambda : (False, ""))
@follows(Up_to_date_task2)
def Up_to_date_task3(infile, outfile):
    pass

@check_if_upToDate (lambda : (False, ""))
@follows(Up_to_date_task3)
```

```
@follows(Up_to_date_task3)
def Up_to_date_final_target(infile, outfile):
    pass

#
# Explicitly specified
#
@check_if_upToDate (lambda : (False, ""))
@follows(Up_to_date_task1)
def Explicitly_specified_task(infile, outfile):
    pass

#
# Tasks to run
#
@follows(Explicitly_specified_task)
def Task_to_run1(infile, outfile):
    pass

@follows(Task_to_run1)
def Task_to_run2(infile, outfile):
    pass

@follows(Task_to_run2)
def Task_to_run3(infile, outfile):
    pass

@check_if_upToDate (lambda : (False, ""))
@follows(Task_to_run2)
def Up_to_date_task_forced_to_rerun(infile, outfile):
    pass

#
# Final target
#
@follows(Up_to_date_task_forced_to_rerun, Task_to_run3)
def Final_target(infile, outfile):
    pass

#
# Ignored downstream
#
@follows(Final_target)
def Downstream_task1_ignored(infile, outfile):
    pass

@follows(Final_target)
def Downstream_task2_ignored(infile, outfile):
    pass
```



```
custom_flow_chart.colour_scheme["Up-to-date Final target"]["fillcolor"] = "#EFC000"
custom_flow_chart.colour_scheme["Up-to-date Final target"]["fontcolor"] = "#00AEEF"
custom_flow_chart.colour_scheme["Up-to-date Final target"]["color"] = "#00AEEF"
custom_flow_chart.colour_scheme["Up-to-date Final target"]["dashed"] = 0

if __name__ == '__main__':
    pipeline_printout_graph (
        open(options.flowchart, "w"),
        # use flowchart file name extension to decide flowchart form
        # e.g. svg, jpg etc.
        os.path.splitext(options.flowchart)[1][1:],
        # final targets
        [Final_target, Up_to_date_final_target],
        # Explicitly specified tasks
        [Explicitly_specified_task],
        # Do we want key legend
        no_key_legend = not options.key_legend_in_graph,
        # Print all the task types whether used or not
        minimal_key_legend = False,
        user_colour_scheme = custom_flow_chart.colour_scheme,
        pipeline_name = "Colour schemes")
```

**OVERVIEW:**

## 2.1 Cheat Sheet

The ruffus module is a lightweight way to add support for running computational pipelines.

Each stage or **task** in a computational pipeline is represented by a python function

Each python function can be called in parallel to run multiple **jobs**.

### 2.1.1 1. Annotate functions with Ruffus decorators

#### Basic

Decorator	Syntax
@follows ( <i>Manual</i> )	<pre>@follows( task1, 'task2' ) @follows( task1, mkdir( 'my/directory/for/results' ) )</pre>
@files ( <i>Manual</i> )	<pre>@files( parameter_list ) @files( parameter_generating_function ) @files( input_file, output_file, other_params, ... )</pre>

## Core

Decorator	Syntax	
<code>@split (Manual)</code>	<code>@split ( tasks_or_file_names, output_files, [extra_parameters,...] )</code>	
<code>@transform (Manual)</code>	<code>@transform ( tasks_or_file_names, suffix(suffix_string), output_pattern, [extra_parameters,...] ) @transform ( tasks_or_file_names, regex(regex_pattern), output_pattern, [extra_parameters,...] )</code>	
<code>@merge (Manual)</code>	<code>@merge (tasks_or_file_names, output, [extra_parameters,...] )</code>	
<code>@posttask (Manual)</code>	<code>@posttask ( signal_task_completion_function ) @posttask (touch_file('task1.completed'))</code>	

See *Decorators* for a complete list of decorators

### 2.1.2 2. Print dependency graph if necessary

- For a graphical flowchart in jpg, svg, dot, png, ps, gif formats:

```
pipeline_printout_graph ( open("flowchart.svg", "w"),  
                         "svg",  
                         list_of_target_tasks)
```

- For a text printout of all jobs

```
pipeline_printout(sys.stdout, list_of_target_tasks)
```

### 2.1.3 3. Run the pipeline

```
pipeline_run(list_of_target_tasks, [list_of_tasks_forced_to_rerun, multiprocess = N_PARALLEL_JOBS])
```

See the *Simple Tutorial* for a quick introduction on how to add support for ruffus. See *Decorators* for more decorators

## 2.2 Pipeline functions

There are only four functions for **Ruffus** pipelines:

- *pipeline\_run* executes a pipeline
- *pipeline\_printout* prints a list of tasks and jobs which will be run in a pipeline
- *pipeline\_printout\_graph* prints a schematic flowchart of pipeline tasks in various graphical formats
- *pipeline\_get\_task\_names* returns a list of all task names in the pipeline

### 2.2.1 *pipeline\_run*

```
pipeline_run ( target_tasks = [], forcedtorun_tasks = [], multiprocess = 1, logger = stderr_logger,  

gnu_make_maximal_rebuild_mode = True, verbose = 1, runtime_data = None, one_second_per_job = True,  

touch_files_only = False)
```

#### Purpose:

Runs all specified pipelined functions if they or any antecedent tasks are incomplete or out-of-date.

#### Example:

```
#  
#   Run task2 whatever its state, and also task1 and antecedents if they are incomplete  
#   Do not log pipeline progress messages to stderr  
#  
pipeline_run([task1, task2], forcedtorun_tasks = [task2], logger = blackhole_logger)
```

#### Parameters:

- ***target\_tasks*** Pipeline functions and any necessary antecedents (specified implicitly or with `@follows`) which should be invoked with the appropriate parameters if they are incomplete or out-of-date.
- ***forcedtorun\_tasks*** Optional. These pipeline functions will be invoked regardless of their state. Any antecedents tasks will also be executed if they are out-of-date or incomplete.
- ***multiprocess*** Optional. The number of processes which should be dedicated to running in parallel independent tasks and jobs within each task. If *multiprocess* is set to 1, the pipeline will execute in the main process.
- ***logger*** For logging messages indicating the progress of the pipeline in terms of tasks and jobs. Defaults to outputting to `sys.stderr`. Setting `logger=blackhole_logger` will prevent any logging output.

- ***gnu\_make\_maximal\_rebuild\_mode***

**Warning:** This is a dangerous option. Use rarely and with caution

Optional parameter governing how **Ruffus** determines which part of the pipeline is out of date and needs to be re-run. If set to `False`, **ruffus** will work back from the `target_tasks` and only execute the pipeline after the first up-to-date tasks that it encounters. For example, if there are four tasks:

```
#  
#   task1 -> task2 -> task3 -> task4 -> task5  
#  
target_tasks = [task5]
```

If `task3()` is up-to-date, then only `task4()` and `task5()` will be run. This will be the case even if `task2()` and `task1()` are incomplete.

This allows you to remove all intermediate results produced by `task1 -> task3`.

- ***verbose*** Optional parameter indicating the verbosity of the messages sent to logger:

```
verbose = 0 : prints nothing
verbose = 1 : logs warnings and tasks which are not up-to-date and which will be run
verbose = 2 : logs doc strings for task functions as well
verbose = 3 : logs job parameters for jobs which are out-of-date
verbose = 4 : logs list of up-to-date tasks but parameters for out-of-date jobs
verbose = 5 : logs parameters for all jobs whether up-to-date or not
verbose = 10: logs messages useful only for debugging ruffus pipeline code
```

`verbose >= 10` are intended for debugging **Ruffus** by the developers and the details are liable to change from release to release

- ***runtime\_data*** Experimental feature for passing data to tasks at run time
- ***one\_second\_per\_job*** By default, **Ruffus** ensures jobs take a minimum of 1 second to complete, to get around coarse grained timestamps in some file systems. This is rarely an issue when many jobs run *in parallel*. If your file system has sub-second time stamps, you can turn off this delay by setting `one_second_per_job` to `False`
- ***touch\_files\_only*** Create or update output files only to simulate the running of the pipeline. Does not invoke real task functions to run jobs. This is most useful to force a pipeline to acknowledge that a particular part is now up-to-date.

This will not work properly if the identities of some files are not known before hand, and depend on run time. In other words, not recommended if `@split` or custom parameter generators are being used.

## 2.2.2 *pipeline\_printout*

**`pipeline_printout`** (`output_stream = sys.stdout`, `target_tasks = []`, `forcedtorun_tasks = []`, `verbose = 1`, `indent = 4`, `gnu_make_maximal_rebuild_mode = True`, `wrap_width = 100`, `runtime_data = None`)

### Purpose:

Prints out all the pipelined functions which will be invoked given specified `target_tasks` without actually running the pipeline. Because this is a simulation, some of the job parameters may be incorrect. For example, the results of a `@split` operation is not predetermined and will only be known after the pipelined function splits up the original data. Parameters of all downstream pipelined functions will be changed depending on this initial operation.

### Example:

```
#  
#   Simulate running task2 whatever its state, and also task1 and antecedents  
#   if they are incomplete  
#   Print out results to STDOUT  
#  
pipeline_printout(sys.stdout, [task1, task2], forcedtorun_tasks = [task2], verbose = 1)
```

### Parameters:

- ***output\_stream*** Where to printout the results of simulating the running of the pipeline.
- ***target\_tasks*** As in `pipeline_run`: Pipeline functions and any necessary antecedents (specified implicitly or with `@follows`) which should be invoked with the appropriate parameters if they are incomplete or out-of-date.

- ***forcedtorun\_tasks*** As in *pipeline\_run*: These pipeline functions will be invoked regardless of their state. Any antecedents tasks will also be executed if they are out-of-date or incomplete.
- ***verbose*** Optional parameter indicating the verbosity of the printout. Please do not expect messages to stay constant between release

```
verbose = 0 : prints nothing
verbose = 1 : logs warnings and tasks which are not up-to-date and which will be run
verbose = 2 : logs doc strings for task functions as well
verbose = 3 : logs job parameters for jobs which are out-of-date
verbose = 4 : logs list of up-to-date tasks but parameters for out-of-date jobs
verbose = 5 : logs parameters for all jobs whether up-to-date or not
verbose = 10: logs messages useful only for debugging ruffus pipeline code
```

- ***indent*** Optional parameter governing the indentation when printing out the component job parameters of each task function.

- ***gnu\_make\_maximal\_rebuild\_mode***

**Warning:** This is a dangerous option. Use rarely and with caution

See explanation in *pipeline\_run*.

- ***wrap\_width*** Optional parameter governing the length of each line before it starts wrapping around.
- ***runtime\_data*** Experimental feature for passing data to tasks at run time

### 2.2.3 *pipeline\_printout\_graph*

```
pipeline_printout_graph (stream, output_format = None, target_tasks = [], forcedtorun_tasks = [], ignore_upstream_of_target = False, skip_update_tasks = False, gnu_make_maximal_rebuild_mode = True, test_all_task_for_update = True, no_key_legend = False, minimal_key_legend = True, user_colour_scheme = None, pipeline_name = "Pipeline", size = (11,8), dpi = 120, runtime_data = None)
```

**Purpose:**

Prints out flowchart of all the pipelined functions which will be invoked given specified target\_tasks without actually running the pipeline.

See *Flowchart colours*

**Example:**

```
pipeline_printout_graph("flowchart.jpg", "jpg", [task1, task16],
                       forcedtorun_tasks = [task2],
                       no_key_legend = True)
```

**Customising appearance:**

The *user\_colour\_scheme* parameter can be used to change flowchart colours. This allows the default *Colour Schemes* to be set. An example of customising flowchart appearance is available (*see code*).

**Parameters:**

- ***stream*** The file or file-like object to which the flowchart should be printed. If a string is provided, it is assumed that this is the name of the output file which will be opened automatically.
- ***output\_format*** If missing, defaults to the extension of the *stream* file name (i.e. jpg for a .jpg)

If the programme `dot` can be found on the execution path, this can be any number of [formats](#) supported by [Graphviz](#), including, for example, jpg, png, pdf, svg etc.

Otherwise, **ruffus** will only output without error in the `dot` format, which is a plain-text graph description language.

- **`target_tasks`** As in `pipeline_run`: Pipeline functions and any necessary antecedents (specified implicitly or with `@follows`) which should be invoked with the appropriate parameters if they are incomplete or out-of-date.
- **`forcedtorun_tasks`** As in `pipeline_run`: These pipeline functions will be invoked regardless of their state. Any antecedent tasks will also be executed if they are out-of-date or incomplete.
- **`draw_vertically`** Draw flowchart in vertical orientation
- **`ignore_upstream_of_target`** Start drawing flowchart from specified target tasks. Do not draw tasks which are downstream (subsequent) to the targets.
- **`ignore_upstream_of_target`** Do not draw up-to-date / completed tasks in the flowchart unless they lie on the execution path of the pipeline.

- **`gnu_make_maximal_rebuild_mode`**

**Warning:** This is a dangerous option. Use rarely and with caution

See explanation in `pipeline_run`.

- **`test_all_task_for_update`**

Indicates whether intermediate tasks are out of date or not. Normally **Ruffus** will stop checking dependent tasks for completion or whether they are out-of-date once it has discovered the maximal extent of the pipeline which has to be run.

For displaying the flow of the pipeline, this is hardly very informative.

- **`no_key_legend`** Do not include key legend explaining the colour scheme of the flowchart.
- **`minimal_key_legend`** Do not include unused task types in key legend.
- **`user_colour_scheme`** Dictionary specifying colour scheme for flowchart

See complete *list of Colour Schemes*.

Colours can be names e.g. "black" or quoted hex e.g. '#F6F4F4' (note extra quotes)

Default values will be used unless specified

key	Subkey	
- 'colour_scheme_index'	index of default colour scheme, 0-7, defaults to 0 unless specified	
- 'Final target' - 'Explicitly specified task' - 'Task to run' - 'Down stream' - 'Up-to-date Final target' - 'Up-to-date task forced to rerun' - 'Up-to-date task' - 'Vicious cycle'	- 'fillcolor' - 'fontcolor' - 'color' - 'dashed' = 0/1	Colours / attributes for each task type
- 'Vicious cycle' - 'Task to run' - 'Up-to-date'	- 'linecolor'	Colours for arrows between tasks
- 'Pipeline'	- 'fontcolor'	Flowchart title colour
- 'Key'	- 'fontcolor' - 'fillcolor'	Legend colours

Example:

Use colour scheme index = 1

```
pipeline_printout_graph ("flowchart.svg", "svg", [final_task],
    user_colour_scheme = {
        "colour_scheme_index" :1,
        "Pipeline"          :{"fontcolor" : "#FF3232",
        "Key"                :{"fontcolor" : "Red",
                                "fillcolor" : "#F6F4F4",
                                "linecolor" : "#0044AA",
                                "fillcolor" : "#EFA03A",
                                "fontcolor" : "black",
                                "dashed"   : 0
        })
    })
```

- ***pipeline\_name*** Specify title for flowchart
- ***size*** Size in inches for flowchart
- ***dpi*** Resolution in dots per inch. Ignored for svg output
- ***runtime\_data*** Experimental feature for passing data to tasks at run time

## 2.2.4 *pipeline\_get\_task\_names*

`pipeline_get_task_names ()`

**Purpose:**

Returns a list of all task names in the pipeline without running the pipeline or checking to see if the tasks are connected correctly

**Example:**

Given:

```
from ruffus import *

@originate([])
def create_data(output_files):
    pass

@transform(create_data, suffix(".txt"), ".task1")
def task1(input_files, output_files):
    pass

@transform(task1, suffix(".task1"), ".task2")
def task2(input_files, output_files):
    pass
```

Produces a list of three task names:

```
>>> pipeline_get_task_names ()
['create_data', 'task1', 'task2']
```

## 2.3 drmaa functions

`drmaa_wrapper` is not exported automatically by ruffus and must be specified explicitly:

```
# imported ruffus.drmaa_wrapper explicitly
from ruffus.drmaa_wrapper import run_job, error_drmaa_job
```

### 2.3.1 `run_job`

`run_job` (*cmd\_str*, *job\_name* = None, *job\_other\_options* = None, *job\_script\_directory* = None, *job\_environment* = None, *working\_directory* = None, *logger* = None, *drmaa\_session* = None, *retain\_job\_scripts* = False, *run\_locally* = False, *output\_files* = None, *touch\_only* = False)

**Purpose:**

`ruffus.drmaa_wrapper.run_job` dispatches a command with arguments to a cluster or Grid Engine node and waits for the command to complete.

It is the semantic equivalent of calling `os.system` or `subprocess.check_output`.

**Example:**

```
from ruffus.drmaa_wrapper import run_job, error_drmaa_job
import drmaa
my_drmaa_session = drmaa.Session()
my_drmaa_session.initialize()

run_job("ls",
        job_name = "test",
        job_other_options="-P mott-flint.prja -q short.qa",
```

```
job_script_directory = "test_dir",
job_environment={ 'BASH_ENV' : '~/.bashrc' },
retain_job_scripts = True, drmaa_session=my_drmaa_session)
run_job("ls",
        job_name = "test",
        job_other_options="-P mott-flint.prja -q short.qa",
        job_script_directory = "test_dir",
        job_environment={ 'BASH_ENV' : '~/.bashrc' },
        retain_job_scripts = True,
        drmaa_session=my_drmaa_session,
        working_directory = "/gpfs1/well/mott-flint/lg/src/oss/ruffus/doc")

#
#   catch exceptions
#
try:
    stdout_res, stderr_res = run_job(cmd,
                                      job_name = job_name,
                                      logger = logger,
                                      drmaa_session = drmaa_session,
                                      run_locally = options.local_run,
                                      job_other_options = get_queue_name())

# relay all the stdout, stderr, drmaa output to diagnose failures
except error_drmaa_job as err:
    raise Exception("\n".join(map(str,
                                   ["Failed to run:",
                                    cmd,
                                    err,
                                    stdout_res,
                                    stderr_res])))

my_drmaa_session.exit()
```

#### Parameters:

- *cmd\_str*

The command which will be run remotely including all parameters

- *job\_name*

A descriptive name for the command. This will be displayed by SGE qstat, for example. Defaults to “ruffus\_job”

- *job\_other\_options*

Other drmaa parameters can be passed verbatim as a string.

Examples for SGE include project name (-P project\_name), parallel environment (-pe parallel\_environ), account (-A account\_string), resource (-l resource=expression), queue name (-q a\_queue\_name), queue priority (-p 15).

These are parameters which you normally need to include when submitting jobs interactively, for example via SGE qsub or SLURM (srun)

- *job\_script\_directory*

The directory where drmaa temporary script files will be found. Defaults to the current working directory.

- *job\_environment*

A dictionary of key / values with environment variables. E.g. "{'BASH\_ENV': '/.bashrc' }"

- *working\_directory*

- Sets the working directory.
- Should be a fully qualified path.
- Defaults to the current working directory.

- *retain\_job\_scripts*

Do not delete temporary script files containing drmaa commands. Useful for debugging, running on the command line directly, and can provide a useful record of the commands.

- *logger*

For logging messages indicating the progress of the pipeline in terms of tasks and jobs. Takes objects with the standard python `logging` module interface.

- *drmaa\_session*

A shared drmaa session created and managed separately.

In the main part of your **Ruffus** pipeline script somewhere there should be code looking like this:

```
#  
#      start shared drmaa session for all jobs / tasks in pipeline  
#  
import drmaa  
drmaa_session = drmaa.Session()  
drmaa_session.initialize()  
  
#  
#      pipeline functions  
#  
  
if __name__ == '__main__':  
    cmdline.run (options, multithread = options.jobs)  
    drmaa_session.exit()
```

- *run\_locally*

Runs commands locally using the standard python `subprocess` module rather than dispatching remotely. This allows scripts to be debugged easily

- *touch\_only*

Create or update *Output files* only to simulate the running of the pipeline. Does not dispatch commands remotely or locally. This is most useful to force a pipeline to acknowledge that a particular part is now up-to-date.

See also: `pipeline_run(touch_files_only=True)`

- *output\_files*

Output files which will be created or updated if *touch\_only* =True

## 2.4 Installation

Ruffus is a lightweight python module for building computational pipelines.

## 2.4.1 The easy way

*Ruffus* is available as an `easy-install`-able package on the Python Package Index.

```
sudo pip install ruffus --upgrade
```

This may also work for older installations

1. Install setuptools:

```
wget peak.telecommunity.com/dist/ez_setup.py  
sudo python ez_setup.py
```

2. Install *Ruffus* automatically:

```
easy_install -U ruffus
```

## 2.4.2 The most up-to-date code:

- Download the latest sources or
- Check out the latest code from Google using git:

```
git clone https://bunbun68@code.google.com/p/ruffus/ .
```

- Bleeding edge Ruffus development takes place on github:

```
git clone git@github.com:bunbun/ruffus.git .
```

- To install after downloading, change to the , type:

```
python ./setup.py install
```

## Graphical flowcharts

*Ruffus* relies on the `dot` programme from `Graphviz` (“Graph visualisation”) to make pretty flowchart representations of your pipelines in multiple graphical formats (e.g. `.png`, `.jpg`). The crossplatform `Graphviz` package can be [downloaded here](#) for Windows, Linux, Macs and Solaris. Some Linux distributions may include prebuilt packages.

**For Fedora, try**

```
yum list 'graphviz*'
```

**For ubuntu / Debian, try**

```
sudo apt-get install graphviz
```

## 2.5 Design & Architecture

The `ruffus` module has the following design goals:

- Simplicity.
- Intuitive
- Lightweight

- Unintrusive
- Flexible/Powerful

Computational pipelines, especially in science, are best thought of in terms of data flowing through successive, dependent stages (**ruffus** calls these *tasks*). Traditionally, files have been used to link pipelined stages together. This means that computational pipelines can be managed using traditional software construction (*build*) systems.

### 2.5.1 *GNU Make*

The grand-daddy of these is UNIX `make`. GNU `make` is ubiquitous in the linux world for installing and compiling software. It has been widely used to build computational pipelines because it supports:

- Stopping and restarting computational processes
- Running multiple, even thousands of jobs in parallel

#### Deficiencies of `make` / `gmake`

However, `make` and GNU `make` use a specialised (domain-specific) language, which has been much criticised because of poor support for modern programming languages features, such as variable scope, pattern matching, debugging. Make scripts require large amounts of often obscure shell scripting and makefiles can quickly become unmaintainable.

### 2.5.2 *Scons, Rake and other Make alternatives*

Many attempts have been made to produce a more modern version of `make`, with less of its historical baggage. These include the Java-based Apache `ant` which is specified in xml.

More interesting are a new breed of build systems whose scripts are written in modern programming languages, rather than a specially-invented “build” specification syntax. These include the Python `scons`, Ruby `rake` and its python port `Smithy`.

The great advantages are that computation pipelines do not need to be artificially parcelled out between (the often second-class) workflow management code, and the logic which does the real computation in the pipeline. It also means that workflow management can use all the standard language and library features, for example, to read in directories, match file names using regular expressions and so on.

**Ruffus** is much like `scons` in that the modern dynamic programming language python is used seamlessly throughout its pipeline scripts.

#### Implicit dependencies: disadvantages of `make` / `scons` / `rake`

Although Python `scons` and Ruby `rake` are in many ways more powerful and easier to use for building software, they are still an imperfect fit to the world of computational pipelines.

This is a result of the way dependencies are specified, an essential part of their design inherited from GNU `make`.

The order of operations in all of these tools is specified in a *declarative* rather than *imperative* manner. This means that the sequence of steps that a build should take are not spelled out explicitly and directly. Instead recipes are provided for turning input files of each type to another.

So, for example, knowing that `a->b, b->c, c->d`, the build system can infer how to get from `a` to `d` by performing the necessary operations in the correct order.

This is immensely powerful for three reasons:

1. The plumbing, such as dependency checking, passing output from one stage to another, are handled automatically by the build system. (This is the whole point!)
2. The same *recipe* can be re-used at different points in the build.
3. Intermediate files do not need to be retained.

Given the automatic inference that  $a \rightarrow b \rightarrow c \rightarrow d$ , we don't need to keep  $b$  and  $c$  files around once  $d$  has been produced.

The disadvantage is that because stages are specified only indirectly, in terms of file name matches, the flow through a complex build or a pipeline can be difficult to trace, and nigh impossible to debug when there are problems.

### Explicit dependencies in *Ruffus*

**Ruffus** takes a different approach. The order of operations is specified explicitly rather than inferred indirectly from the input and output types. So, for example, we would explicitly specify three successive and linked operations  $a \rightarrow b$ ,  $b \rightarrow c$ ,  $c \rightarrow d$ . The build system knows that the operations always proceed in this order.

Looking at a **Ruffus** script, it is always clear immediately what is the succession of computational steps which will be taken.

**Ruffus** values clarity over syntactic cleverness.

### Static dependencies: What *make* / *scons* / *rake* can't do (easily)

GNU *make*, *scons* and *rake* work by infer a static dependency (diacyclic) graph between all the files which are used by a computational pipeline. These tools locate the target that they are supposed to build and work backward through the dependency graph from that target, rebuilding anything that is out of date. This is perfect for building software, where the list of files data files can be computed **statically** at the beginning of the build.

This is not ideal matches for scientific computational pipelines because:

- Though the *stages* of a pipeline (i.e. *compile* or *DNA alignment*) are invariably well-specified in advance, the number of operations (*jobs*) involved at each stage may not be.
- A common approach is to break up large data sets into manageable chunks which can be operated on in parallel in computational clusters or farms (See [embarassingly parallel problems](#)).  
This means that the number of parallel operations or jobs varies with the data (the number of manageable chunks), and dependency trees cannot be calculated statically beforehand.

Computational pipelines require **dynamic** dependencies which are not calculated up-front, but at each stage of the pipeline

This is a *known* issue with traditional build systems each of which has partial strategies to work around this problem:

- *gmake* always builds the dependencies when first invoked, so dynamic dependencies require (complex!) recursive calls to *gmake*
- *Rake* dependencies unknown prior to running tasks.

- Scons: Using a Source Generator to Add Targets Dynamically

**Ruffus** explicitly and straightforwardly handles tasks which produce an indeterminate (i.e. runtime dependent) number of output, using its `@split`, `@transform`, `merge` function annotations.

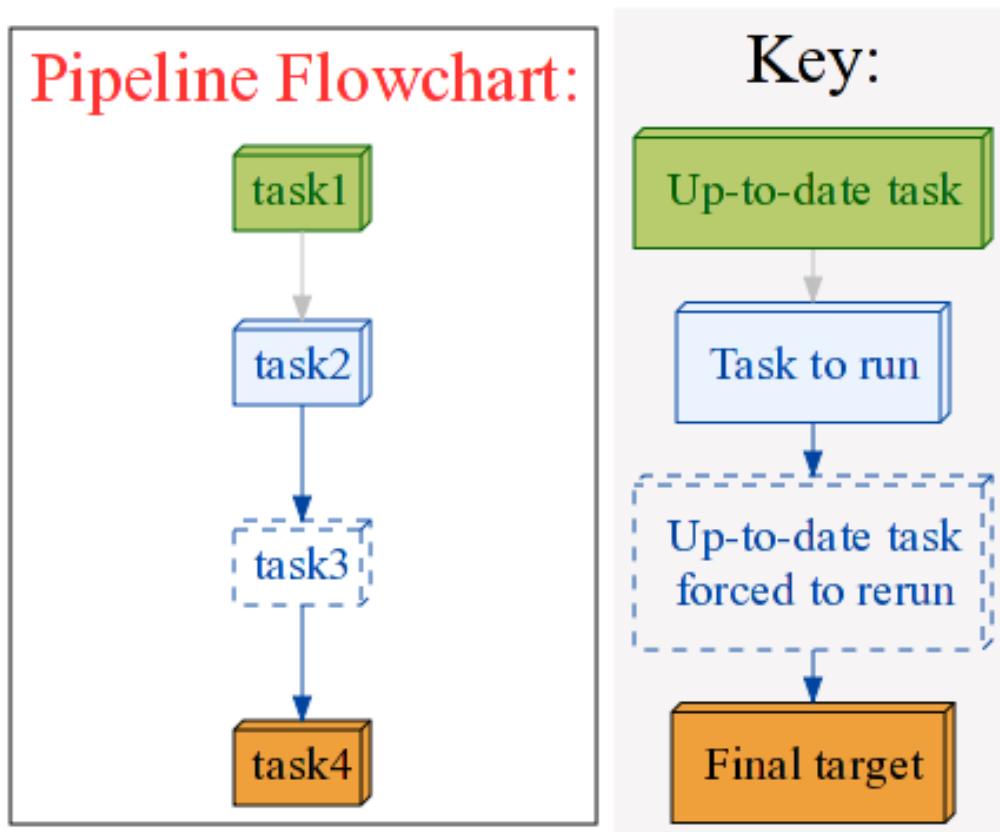
### 2.5.3 Managing pipelines stage-by-stage using Ruffus

**Ruffus** manages pipeline stages directly.

1. The computational operations for each stage of the pipeline are written by you, in separate python functions.

(These correspond to gmake pattern rules)

2. The dependencies between pipeline stages (python functions) are specified up-front.  
These can be displayed as a flow chart.



3. **Ruffus** makes sure pipeline stage functions are called in the right order, with the right parameters, running in parallel using multiprocessing if necessary.
4. Data file timestamps can be used to automatically determine if all or any parts of the pipeline are out-of-date and need to be rerun.
5. Separate pipeline stages, and operations within each pipeline stage, can be run in parallel provided they are not inter-dependent.

Another way of looking at this is that **ruffus** re-constructs datafile dependencies dynamically on-the-fly when it gets to each stage of the pipeline, giving much more flexibility.

### Disadvantages of the Ruffus design

Are there any disadvantages to this trade-off for additional clarity?

1. Each pipeline stage needs to take the right input and output. For example if we specified the steps in the wrong order: a->b, c->d, b->c, then no useful output would be produced.
2. We cannot re-use the same recipes in different parts of the pipeline
3. Intermediate files need to be retained.

In our experience, it is always obvious when pipeline operations are in the wrong order, precisely because the order of computation is the very essence of the design of each pipeline. Ruffus produces extra diagnostics when no output is created in a pipeline stage (usually happens for incorrectly specified regular expressions.)

Re-use of recipes is as simple as an extra call to common function code.

Finally, some users have proposed future enhancements to **Ruffus** to handle unnecessary temporary / intermediate files.

### 2.5.4 Alternatives to Ruffus

A comparison of more make-like tools is available from [Ian Holmes' group](#).

Build systems include:

- [GNU make](#)
- [scons](#)
- [ant](#)
- [rake](#)

There are also complete workload management systems such as Condor. Various bioinformatics pipelines are also available, including that used by the leading genome annotation website Ensembl, Pegasys, GPIPE, Taverna, Wildfire, MOWserv, Triana, Cyrille2 etc. These all are either hardwired to specific databases, and tasks, or have steep learning curves for both the scientist/developer and the IT system administrators.

**Ruffus** is designed to be lightweight and unintrusive enough to use for writing pipelines with just 10 lines of code.

See also:

#### Bioinformatics workload management systems

**Condor:** <http://www.cs.wisc.edu/condor/description.html>

**Ensembl Analysis pipeline:** <http://www.ncbi.nlm.nih.gov/pubmed/15123589>

**Pegasys:** <http://www.ncbi.nlm.nih.gov/pubmed/15096276>

**GPIPE:** <http://www.biomedcentral.com/pubmed/15096276>

**Taverna:** <http://www.ncbi.nlm.nih.gov/pubmed/15201187>

**Wildfire:** <http://www.biomedcentral.com/pubmed/15788106>

**MOWserv:** <http://www.biomedcentral.com/pubmed/16257987>

**Triana:** <http://dx.doi.org/10.1007/s10723-005-9007-3>

**Cyrille2:** <http://www.biomedcentral.com/1471-2105/9/96>

## Acknowledgements

- Bruce Eckel's insightful article on [A Decorator Based Build System](#) was the obvious inspiration for the use of decorators in *Ruffus*.

**The rest of the *Ruffus* takes uses a different approach. In particular:**

1. *Ruffus* uses task-based not file-based dependencies
2. *Ruffus* tries to have minimal impact on the functions it decorates.

Bruce Eckel's design wraps functions in "rule" objects.

*Ruffus* tasks are added as attributes of the functions which can be still be called normally. This is how *Ruffus* decorators can be layered in any order onto the same task.

- Languages like c++ and Java would probably use a "mixin" approach. Python's easy support for reflection and function references, as well as the necessity of marshalling over process boundaries, dictated the internal architecture of *Ruffus*.
- The [Boost Graph library](#) for text book implementations of directed graph traversals.
- [Graphviz](#). Just works. Wonderful.
- Andreas Heger, Christoffer Nellåker and Grant Belgard for driving Ruffus towards ever simpler syntax.

## 2.6 Major Features added to Ruffus

### 2.6.1 version 2.4.2

- BUG FIX: Output producing wild cards was not saved in the checksum files!!!
- Added `pipeline_get_task_names(...)` which returns all task name as a list of strings
- **Reorganised verbosity for `pipeline_printout` and `pipeline_run`**
  - level 0 : nothing
  - level 1 : Out-of-date Tasks (names and warnings)
  - level 2 : All Tasks (including any task function docstrings)
  - level 3 : Out-of-date Jobs in Out-of-date Tasks, no explanation
  - level 4 : Out-of-date Jobs in Out-of-date Tasks, with explanation
  - level 5 : All Jobs in Out-of-date Tasks, (include only list of up-to-date tasks)
  - level 6 : All jobs in All Tasks whether out of date or not
  - level 10: logs messages useful only for debugging ruffus pipeline code
- Need to update docs

- test/test\_verbosity.py
- path\_verbosity = (default = 2)
- 0) the full path 1-N) N levels of subpath,  
for “what/is/this.txt” N = 1 “this.txt” N = 2 “is/this.txt” N >=3 “what/is/this.txt”

**-N) As above but with the input/output parameter chopped off after 40 letters, and ending in “...”**

Getting the Nth levels of a path use code from ruffus.ruffus\_utility

from ruffus.ruffus\_utility import \* for aa in range(10):

```
print get_nth_nested_level_of_path (“/test/this/now/or/not.txt”, aa)
```

May 29, 2014 Delete comment Project Member #2 bunbun68 1) pipeline\_printout forwards printing to  
\_task.printout 2) pipeline\_run needs to borrow code from pipeline\_printout to print up to date tasks in-  
cluding their jobs 3) See file\_name\_parameters.py::get\_readable\_path\_str()

## 2.6.2 version 2.4.1

- Breaking changes to drmaa API suggested by Bernie Pope to ensure portability across different drmaa imple-  
mentations (SGE, SLURM etc.)

## 2.6.3 version 2.4

### Additions to ruffus namespace

- *formatter()* (*syntax*)
- *originate()* (*syntax*)
- *subdivide()* (*syntax*)

### Installation: use pip

```
sudo pip install ruffus --upgrade
```

#### 1) Command Line support

The optional Ruffus.cmdline module provides support for a set of common command line arguments  
which make writing *Ruffus* pipelines much more pleasant. See *manual*

#### 2) Check pointing

- Contributed by **Jake Biesinger**
- See *Manual*
- Uses a fault resistant sqlite database file to log i/o files, and additional checksums
- defaults to checking file timestamps stored in the current directory  
(ruffus\_utility.RUFFUS\_HISTORY\_FILE = ‘.ruffus\_history.sqlite’)
- *pipeline\_run(..., checksum\_level = N, ...)*

- level 0 = CHECKSUM\_FILE\_TIMESTAMPS : Classic mode. Use only file timestamps (no checksum file will be created)
  - level 1 = CHECKSUM\_HISTORY\_TIMESTAMPS : Also store timestamps in a database after successful job completion
  - level 2 = CHECKSUM\_FUNCTIONS : As above, plus a checksum of the pipeline function body
  - level 3 = CHECKSUM\_FUNCTIONS\_AND\_PARAMS : As above, plus a checksum of the pipeline function default arguments and the additional arguments passed in by task decorators
  - defaults to level 1
- Can speed up trivial tasks: Previously Ruffus always added an extra 1 second pause between tasks to guard against file systems (Ext3, FAT, some NFS) with low timestamp granularity.

### 3) ***subdivide()*** (*syntax*)

- 
- Take a list of input jobs (like `@transform`) but further splits each into multiple jobs, i.e. it is a **many->even more** relationship
- synonym for the deprecated `@split(..., regex(), ...)`

### 4) ***mkdir()*** (*syntax*) with ***formatter()*, *suffix()* and *regex()***

- allows directories to be created depending on runtime parameters or the output of previous tasks
- behaves just like `@transform` but with its own (internal) function which does the actual work of making a directory
- Previous behavior is retained:`mkdir` continues to work seamlessly inside `@follows`

### 5) ***originate()*** (*syntax*)

- Generates output files without dependencies from scratch (*ex nihilo!*)
- For first step in a pipeline
- Task function obviously only takes output and not input parameters. (There *are* no inputs!)
- synonym for `@split(None,...)`
- See *Summary / Manual*

### 6) New flexible ***formatter()*** (*syntax*) alternative to ***regex()* & *suffix()***

- Easy manipulation of path subcomponents in the style of `os.path.split()`
- Regular expressions are no longer necessary for path manipulation
- Familiar python syntax
- Optional regular expression matches
- Can refer to any in the list of N input files (not only the first file as for `regex(...)`)
- Can even refer to individual letters within a match

## 7) Combinatorics (all vs. all decorators)

- `@product` (See `itertools.product`)
- `@permutations` (See `itertools.permutations`)
- `@combinations` (See `itertools.combinations`)
- `@combinations_with_replacement` (See `itertools.combinations_with_replacement`)
- in optional `combinatorics` module
- Only `formatter()` provides the necessary flexibility to construct the output. (`suffix()` and `regex()` are not supported.)
- See *Summary / Manual*

## 8) drmaa support and multithreading:

- `ruffus.drmaa_wrapper.run_job()` (`syntax`)
- Optional helper module allows jobs to dispatch work to a computational cluster and wait until it completes.
- Requires `multithread` rather than `multiprocess`

## 9) `pipeline_run(...)` and exceptions

See *Manual*

- Optionally terminate pipeline after first exception
- Display exceptions without delay

## 10) Miscellaneous

Better error messages for `formatter()`, `suffix()` and `regex()` for `pipeline_printout(..., verbose >= 3)`

- Error messages for showing mismatching regular expression and offending file name
- Wrong capture group names or out of range indices will raise informative Exception

## 2.6.4 version 2.3

- **`@active_if` turns off tasks at runtime** The Design and initial implementation were contributed by Jacob Biesinger

Takes one or more parameters which can be either booleans or functions or callable objects which return True / False:

```
run_if_true_1 = True
run_if_true_2 = False

@active_if(run_if_true, lambda: run_if_true_2)
def this_task_might_be_inactive():
    pass
```

The expressions inside @active\_if are evaluated each time pipeline\_run, pipeline\_printout or pipeline\_printout\_graph is called.

Dormant tasks behave as if they are up to date and have no output.

- **Command line parsing**

- Supports both argparse (python 2.7) and optparse (python 2.6):
- Ruffus cmdline module is optional.
- See *manual*

- **Optionally terminate pipeline after first exception** To have all exceptions interrupt immediately:

```
pipeline_run(..., exceptions_terminate_immediately = True)
```

By default ruffus accumulates NN errors before interrupting the pipeline prematurely. NN is the specified parallelism for pipeline\_run(..., multiprocess = NN).

Otherwise, a pipeline will only be interrupted immediately if exceptions of type ruffus.JobSignalledBreak are thrown.

- **Display exceptions without delay**

By default, Ruffus re-throws exceptions in ensemble after pipeline termination.

To see exceptions as they occur:

```
pipeline_run(..., log_exceptions = True)
```

logger.error(...) will be invoked with the string representation of the each exception, and associated stack trace.

The default logger prints to sys.stderr, but this can be changed to any class from the logging module or compatible object via pipeline\_run(..., logger = ???)

- **Improved pipeline\_printout()**

- @split operations now show the 1->many output in pipeline\_printout

This make it clearer that @split is creating multiple output parameters (rather than a single output parameter consisting of a list):

```
Task = split_animals
      Job = [None
              -> cows
              -> horses
              -> pigs
              , any_extra_parameters]
```

- File date and time are displayed in human readable form and out of date files are flagged with asterisks.

## 2.6.5 version 2.2

- Simplifying @transform syntax with suffix(...)

Regular expressions within ruffus are very powerful, and can allow files to be moved from one directory to another and renamed at will.

However, using consistent file extensions and @transform(..., suffix(...)) makes the code much simpler and easier to read.

Previously, `suffix(...)` did not cooperate well with `inputs(...)`. For example, finding the corresponding header file (“.h”) for the matching input required a complicated `regex(...)` regular expression and `input(...)`. This simple case, e.g. matching “something.c” with “something.h”, is now much easier in Ruffus.

#### For example:

```
source_files = ["something.c", "more_code.c"]
@transform(source_files, suffix(".c"), add_inputs(r"\1.h", "common.h"), ".o")
def compile(input_files, output_file):
    ( source_file,
      header_file,
      common_header) = input_files
    # call compiler to make object file
```

This is equivalent to calling:

```
compile(["something.c", "something.h", "common.h"], "something.o")
compile(["more_code.c", "more_code.h", "common.h"], "more_code.o")
```

The `\1` matches everything *but* the suffix and will be applied to both `glob`s and file names.

For simplicity and compatibility with previous versions, there is always an implied `r"1"` before the output parameters. I.e. output parameters strings are *always* substituted.

- Tasks and `glob` in `inputs(...)` and `add_inputs(...)`

`glob`s and tasks can be added as the prerequisites / input files using `inputs(...)` and `add_inputs(...)`. `glob` expansions will take place when the task is run.

- Advanced form of `@split` with `regex`:

The standard `@split` divided one set of inputs into multiple outputs (the number of which can be determined at runtime).

This is a `one->many` operation.

An advanced form of `@split` has been added which can split each of several files further.

In other words, this is a `many->"many more"` operation.

#### For example, given three starting files:

```
original_files = ["original_0.file",
                 "original_1.file",
                 "original_2.file"]
```

We can split each into its own set of sub-sections:

```
@split(original_files,
       regex(r"starting_(\d+).fa"),
       r"files.split.\1.*.fa"
       r"\1")
def split_files(input_file, output_files, original_index):
    """
        Code to split each input_file
        "original_0.file" -> "files.split.0.*.fa"
        "original_1.file" -> "files.split.1.*.fa"
        "original_2.file" -> "files.split.2.*.fa"
    """
```

This is, conceptually, the reverse of the @collate(...) decorator

- Ruffus will complain about unescaped regular expression special characters:

Ruffus uses “\1” and “\2” in regular expression substitutions. Even seasoned python users may not remember that these have to be ‘escaped’ in strings. The best option is to use ‘raw’ python strings e.g.

```
r"\1_substitutes\2correctly\3four\4times"
```

Ruffus will throw an exception if it sees an unescaped “\1” or “\2” in a file name, which should catch most of these bugs.

- Prettier output from *pipeline\_printout\_graph*

Changed to nicer colours, symbols etc. for a more professional look. @split and @merge tasks now look different from @transform. Colours, size and resolution are now fully customisable:

```
pipeline_printout_graph( #...
    user_colour_scheme = {
        "colour_scheme_index":1,
        "Task to run" : {"fillcolor":"blue"},
        pipeline_name : "My flowchart",
        size          : (11,8),
        dpi           : 120})
```

An SVG bug in firefox has been worked around so that font size are displayed correctly.

## 2.6.6 version 2.1.1

- **@transform(..., add\_inputs(...))** `add_inputs(...)` allows the addition of extra input dependencies / parameters for each job.

**Unlike `inputs(...)`, the original input parameter is retained:**

```
from ruffus import *
@transform(["a.input", "b.input"], suffix(".input"), add_inputs("just.1.more","just.2.more"))
def task(i, o):
    ""
```

**Produces:**

```
Job = [[a.input, just.1.more, just.2.more] ->a.output]
Job = [[b.input, just.1.more, just.2.more] ->b.output]
```

Like `inputs`, `add_inputs` accepts strings, tasks and glob s This minor syntactic change promises add much clarity to Ruffus code. `add_inputs()` is available for `@transform`, `@collate` and `@split`

## 2.6.7 version 2.1.0

- **@jobs\_limit** Some tasks are resource intensive and too many jobs should not be run at the same time. Examples include disk intensive operations such as unzipping, or downloading from FTP sites.

Adding:

```
@jobs_limit(4)
@transform(new_data_list, suffix(".big_data.gz"), ".big_data")
def unzip(i, o):
    "unzip code goes here"
```

would limit the unzip operation to 4 jobs at a time, even if the rest of the pipeline runs highly in parallel.

(Thanks to Rob Young for suggesting this.)

## 2.6.8 version 2.0.10

- **touch\_files\_only** option for **pipeline\_run**

When the pipeline runs, task functions will not be run. Instead, the output files for each job (in each task) will be `touch-ed` if necessary. This can be useful for simulating a pipeline run so that all files look as if they are up-to-date.

Caveats:

- This may not work correctly where output files are only determined at runtime, e.g. with `@split`
  - Only the output from pipelined jobs which are currently out-of-date will be `touch-ed`. In other words, the pipeline runs *as normal*, the only difference is that the output files are `touch-ed` instead of being created by the python task functions which would otherwise have been called.
- Parameter substitution for **inputs(...)**

The **inputs(...)** parameter in `@transform`, `@collate` can now take tasks and `glob`s, and these will be expanded appropriately (after regular expression replacement).

For example:

```
@transform("dir/a.input", regex(r"(.*)\/(.+)\\.input"),
           inputs((r"\1/\2.other", r"\1/*.more")), r"elsewhere/\2.output")
def task1(i, o):
    """
    Some pipeline task
    """
```

Is equivalent to calling:

```
task1(("dir/a.other", "dir/1.more", "dir/2.more"), "elsewhere/a.output")
```

Here:

```
r"\1/*.more"
```

is first converted to:

```
r"dir/*.more"
```

which matches:

```
"dir/1.more"
"dir/2.more"
```

## 2.6.9 version 2.0.9

- Better display of logging output

- Advanced form of **@split** This is an experimental feature.

Hitherto, **@split** only takes 1 set of input (tasks/files/glob s) and split these into an indeterminate number of output.

This is a one->many operation.

Sometimes it is desirable to take multiple input files, and split each of them further.

This is a many->many (more) operation.

It is possible to hack something together using **@transform** but downstream tasks would not aware that each job in **@transform** produces multiple outputs (rather than one input, one output per job).

The syntax looks like:

```
@split(get_files, regex(r"(.).original"), r"\1.*.split")
def split_files(i, o):
    pass
```

If `get_files()` returned `A.original`, `B.original` and `C.original`, `split_files()` might lead to the following operations:

```
A.original
    -> A.1.original
    -> A.2.original
    -> A.3.original
B.original
    -> B.1.original
    -> B.2.original
C.original
    -> C.1.original
    -> C.2.original
    -> C.3.original
    -> C.4.original
    -> C.5.original
```

Note that each input (`A/B/C.original`) can produce a number of output, the exact number of which does not have to be pre-determined. This is similar to **@split**

Tasks following `split_files` will have ten inputs corresponding to each of the output from `split_files`.

If **@transform** was used instead of **@split**, then tasks following `split_files` would only have 3 inputs.

## 2.6.10 version 2.0.8

- File names can be in unicode
- File systems with 1 second timestamp granularity no longer cause problems.

## 2.6.11 version 2.0.2

- Much prettier /useful output from `pipeline_printout`
- New tutorial / manual

## 2.6.12 version 2.0

- Revamped documentation:
  - Rewritten tutorial
  - Comprehensive manual
  - New syntax help
- Major redesign. New decorators include
  - `@split`
  - `@transform`
  - `@merge`
  - `@collate`
- Major redesign. Decorator *inputs* can mix
  - Output from previous tasks
  - *glob* patterns e.g. `*.txt`
  - Files names
  - Any other data type

## 2.6.13 version 1.1.4

Tasks can get their input by automatically chaining to the output from one or more parent tasks using `@files_re`

## 2.6.14 version 1.0.7

Added `proxy_logger` module for accessing a shared log across multiple jobs in different processes.

## 2.6.15 version 1.0

Initial Release in Oxford

## 2.7 Fixed Bugs

Full list at “[Latest Changes](#) wiki entry”

## 2.8 Future plans for *Ruffus*:

### 2.8.1 Update documentation

## 2.9 Left to do:

I would appreciate feedback and help on all these issues and where next to take *ruffus*.

Please write to me ( ruffus\_lib at llew.org.uk) or join the project.

Some of these proposals are well-fleshed-out:

- *Clean up*
- *(Plug-in) File Dependency Checking via MD5 or Databases*

Others require some more user feedback about semantics:

- *Harvesting return values from jobs*

Some issues are do-able but difficult and I don't have the experience:

- *Run jobs on remote (clustered) processes via SGE/Hadoop*

## 2.9.1 Exceptions

The current behaviour is to continue executing all the jobs currently in progress when an exception is thrown (See the *manual*).

A.H. has suggested that

- Exceptions should be displayed early
- Ctrl-C should not leave dangling jobs
- As an option, Ruffus should try to keep running as far as possible (i.e. ignoring downstream tasks)

## 2.9.2 Clean up

The plan is to store the files and directories created via a standard interface.

The placeholders for this are a function call `register_cleanup`.

Jobs can specify the files they created and which need to be deleted by returning a list of file names from the job function.

So:

```
raise Exception = Error

return False = halt pipeline now

return string / list of strings = cleanup files/directories later

return anything else = ignored
```

The cleanup file/directory store interface can be connected to a text file or a database.

The cleanup function would look like this:

```
pipeline_cleanup(cleanup_log("../cleanup.log"), [instance ="october19th"])
pipeline_cleanup(cleanup_msql_db("user", "password", "hash_record_table"))
```

The parameters for where and how to store the list of created files could be similarly passed to `pipeline_run` as an extra parameter:

```
pipeline_run(cleanup_log("../cleanup.log"), [instance ="october19th"])
pipeline_run(cleanup_msql_db("user", "password", "hash_record_table"))
```

where `cleanup_log` and `cleanup_msql_db` are classes which have functions for

1. storing file
2. retrieving file
3. clearing entries
  - Files would be deleted in reverse order, and directories after files.
  - By default, only empty directories would be removed.  
But this could be changed with a `--forced_remove_dir` option
  - An `--remove_empty_parent_directories` option would be supported by `os.removedirs(path)`.

### 2.9.3 (Plug-in) File Dependency Checking via MD5 or Databases

So that MD5 / a database can be used instead of coarse-grained file modification times.

As always, the design is a compromise between flexibility and easy of use.

The user can already write their own file dependency checking function and supply this:

```
@check_if_upToDate(check_md5_func)
@files(io_files)
def task_func (input_file, output_file):
    pass
```

The question is can we

1. supply a `check_md5()` function
2. allow the whole pipeline to use this.

Most probably we need an extra parameter somewhere:

```
pipeline_run(md5_hash_database = "current/location/files.md5")
```

There is prior art on this in scons.

If we use a custom object/function, can we use orthogonal syntax for

1. disk modifications times,
2. md5 hashes saved to a file,
3. md5 hashes saved to a database?

```
pipeline_run(file_up_to_date_lookup = md5_hash_file("current/location/files.md5"))
pipeline_run(file_up_to_date_lookup = mysql_hash_store("user", "password", "hash_record_table"))
```

where `md5_hash_file` and `mysql_hash_store` are objects which have get/set functions for looking up modification times from file names.

Of course that allows you to fake the whole process and not even use real files...

### 2.9.4 Remove intermediate files

Often large intermediate files are produced in the middle of a pipeline which could be removed. However, their absence would cause the pipeline to appear out of date. What is the best way to solve this?

In gmake, all intermediate files which are not marked `.PRECIOUS` are deleted.

**We do not want to manually mark intermediate files for several reasons:**

- The syntax would be horrible and clunky
- The gmake distinction between `implicit` and `explicit` rules is not one we would like to impose on Ruffus
- Gmake uses statically determined (DAG) dependency trees so it is quite natural and easy to prune intermediate paths

Our preferred solution should impose little to no semantic load on Ruffus, i.e. it should not make it more complex / difficult to use. There are several alternatives we are considering:

1. Have an `update` mode in which `pipeline_run` would ignore missing files and only run tasks with existing, out-of-date files.
2. Optionally ignore all out-of-date dependencies beyond a specified point in the pipeline
3. Add a decorator to flag sections of the pipeline where intermediate files can be removed

Option (1) is rather unnerving because it makes inadvertent errors difficult to detect.

Option (2) involves relying on the user of a script to remember the correct chain of dependencies in often complicated pipelines. It would be advised to keep a flowchart to hand. Again, the chances of error are much greater.

Option (3) springs from the observation by Andreas Heger that parts of a pipeline with disposable intermediate files can usually be encapsulated as an autonomous section. Within this subpipeline, all is well provided that the outputs of the last task are complete and up-to-date with reference to the inputs of the first task. Intermediate files could be removed with impunity.

The suggestion is that these autonomous subpipelines could be marked out using the Ruffus decorator syntax:

```
#  
#    First task in autonomous subpipeline  
#  
@files("who.isit", "its.me")  
def first_task(*args):  
    pass  
  
#  
#    Several intermediate tasks  
#  
@transform(subpipeline_task1, suffix(".me"), ".her")  
def task2_etc(*args):  
    pass  
  
#  
#    Final task  
#  
@sub_pipeline(subpipeline_task1)  
@transform(subpipeline_task1, suffix(".her"), ".you")  
def final_task(*args):  
    pass
```

`@sub_pipeline` marks out all tasks between `first_task` and `final_task` and intermediate files such as `"its.me"`, `"its.her"` can be deleted. The pipeline will only run if `"its.you"` is missing or out-of-date compared with `"who.isit"`.

Over the next few Ruffus releases we will see if this is a good design, and whether better keyword can be found than `@sub_pipeline` (candidates include `@shortcut` and `@intermediate`)

## 2.9.5 Extra signalling before and after each task and job

@pretask(custom\_func) @prejob(custom\_func) @postjob(custom\_func)

@pretask would be run in the master process while @prejob / @postjob would be run in the child processes (if any).

## 2.9.6 SQL hooks

See above.

I have no experience with systems which link to SQL. What would people want from such a feature?

Ian Holmes?

## 2.9.7 Return values

Is it a good idea to allow jobs to pass back calculated values?

This requires trivial modifications to run\_pooled\_job\_without\_exceptions

The most useful thing would be to associate job parameters with results.

What should be the syntax for getting the results back?

## 2.9.8 Run jobs on remote (clustered) processes via SGE/Hadoop

Can we run jobs on remote processes using SGE / Hadoop?

Can we abstract all job management using drmaa?

Python examples at [http://gridengine.sunsource.net/howto/drmaa\\_python.html](http://gridengine.sunsource.net/howto/drmaa_python.html)

### SGE

Look at Qmake execution model:

#### 1) SGE nodes are taken over completely

See last example in [multiprocessing](#) for creating a distributed queue.

We would use qrsh instead of ssh. The size of the pool would be the (maximum) number of jobs

Advantages:

- Simple to implement
- Efficient

Disadvantages:

- Other users might not appreciate having python jobs taking over the nodes for a protracted length of time
- We would not be able to use SGE to view / manage jobs

## 2) Start a qrsh per job

Advantages:

- jobs look like any other SGE task

Disadvantages:

- Slower. Overheads might be high.
- We might have to create a new pool per task
- If we maintain an empty pool, and then dynamically attach processes, the code might be difficult to write (may not fit into the multiprocessing way of doing things / race-conditions etc.)

## Hadoop

Can anyone help me with this / have any experience?

## 2.10 In progress: Refactoring Ruffus Docs

Remember to cite Jake Biesinger and see if he is interested to be a co-author if we ever resubmit the drastically changed version...

## 2.11 Future / Planned Improvements to Ruffus

### 2.11.1 Todo: Bioinformatics example to end all examples

#### Uses

- @product
- @subdivide
- @transform
- @collate
- @merge

### 2.11.2 Todo: Running python code (task functions) transparently on remote cluster nodes

Wait until next release.

Will bump Ruffus to v.3.0 if can run python jobs transparently on a cluster!

abstract out `task.run_pooled_job_without_exceptions()` as a function which can be supplied to `pipeline_run`

Common “job” interface:

- marshalled arguments
- marshalled function

- submission timestamp

#### Returns

- completion timestamp
  - returned values
  - exception
1. Full version use libpythongrid? \* Christian Widmer <ckwidmer@gmail.com> \* Cheng Soon Ong <chengsoon.ong@unimelb.edu.au> \* <https://code.google.com/p/pythongrid/source/browse/#git%2Fpythongrid> \* Probably not good to base Ruffus entirely on libpythongrid to minimise dependencies, the use of sophisticated configuration policies etc.
  2. Start with light-weight file-based protocol \* specify where the scripts should live \* use drmaa to start jobs \* have executable ruffus module which knows how to load deserialise (unmarshall) function / parameters from disk. This would be what drmaa starts up, given the marshalled data as an argument \* time stamp \* “heart beat” to check that the job is still running
  3. Next step: socket-based protocol \* use specified master port in ruffus script \* start remote processes using drmaa \* child receives marshalled data and the address::port in the ruffus script (head node) to initiate hand shake or die \* process recycling: run successive jobs on the same remote process for reduced overhead, until exceeds max number of jobs on the same process, min/max time on the same process \* resubmit if die (Don’t do sophisticated stuff like libpythongrid).

### 2.11.3 Mark input strings as non-file names, and add support for dynamically returned parameters

1. Use indicator object like “output\_from”
2. What is a good name?
3. They will still participate in suffix, formatter and regex replacement

Bernie Pope suggests that we should generalise this:

If any object in the input parameters is a (non-list/tuple) class instance, check (getattr) whether it has a “ruffus\_params()” function. If it does, call it to obtain a list which is substituted in place. If there are string nested within, these will take part in Ruffus string substitution.

“output\_from” would be a simple wrapper which returns the internal string via ruffus\_params()

```
class output_from (object):  
    def __init__(self, str):  
        self.str = str  
    def ruffus_params(self):  
        return [self.str]
```

Returning a list should be like wildcards and should not introduce an unnecessary level of indirection for output parameters, i.e. suffix(“.txt”) or formatter() / “{basename[0]}” should work.

Check!

### 2.11.4 Allow “extra” parameters to be used in output substitution

Formatter substitution can refer to the original elements in the input and extra parameters (without converting them to strings either). This refers to the original (nested) data structure.

This will allow normal python datatypes to be handed down and slipstreamed into a pipeline more easily.

The syntax would use Ruffus (> version 2.4) formatter:

```
@transform( ..., formatter(), ["{EXTRAS[0][1][3]}", "[INPUTS[1][2]]"], ...)
```

EXTRA and INPUTS indicate that we are referring to the input and extra parameters.

These are the full (nested) parameters in all their original form. In the case of the input parameters, this obvious depends on the decorator, so

```
@transform(["a.text", [1, "b.text"]], formatter(), "{INPUTS[0][0]}")
```

would give

```
job #1
    input == "a.text"
    output == "a"

job #2
    input == [1, "b.text"]
    output == 1
```

The entire string must consist of INPUTS or EXTRAS followed by optionally N levels of square brackets.  
i.e. They must match “(INPUTS|EXTRAS)([d+])<sup>N</sup>”

No string conversion takes place.

## 2.11.5 Refactor verbosity levels

Verbosity levels for pipeline\_printout and pipeline\_run do not seem to be synchronised. It is not clear what exactly increasing verbosity does at each level. What is more, different things seem to happen differently at run time and print\_out.

```
verbosity=
    1. Out-of-date Tasks
    2. All Tasks
    3. Out-of-date Jobs in Out-of-date Tasks
    4. All Jobs in Out-of-date Tasks
    5. All jobs in All Tasks whether out of date or not
path_verbosity =
    (default = 2)
    0) the full path 1-N) N levels of subpath,
```

```
        for "what/is/this.txt" N = 1 "this.txt" N = 2 "is/this.txt" N >=3 "what/is/this.txt"
    -N) As above but with the input/output parameter chopped off after 40 letters, and ending in
        "..."
```

Getting the Nth levels of a path use code from ruffus.ruffus\_utility

```
from ruffus.ruffus_utility import *
for aa in range(10):
    print get_nth_nested_level_of_path ("/test/this/now/or/not.txt", aa)
```

1. pipeline\_printout forwards printing to \_task.printout

2. pipeline\_run needs to borrow code from pipeline\_printout to print up to date tasks including their jobs
3. See file\_name\_parameters.py::get\_readable\_path\_str()

## 2.11.6 New decorators

### Planned: @split / @subdivide

Can't see how we can stop using wild cards but at least if we return output strings in the task functions, we don't include extraneous files which were not created in the pipeline but which just happened to match the wild card in the function.

We should check whether we have ever run the function before, and if we have to also only check the files which we generated last time...

### Planned: @originate

Each (serial) invocation returns lists of output parameters until returns None. (Empty list = continue, None = break).

### Planned: @recombine

Like @collate but automatically regroups jobs which were a result of a previous @subdivide / @split (even after intervening @transform)

This is the only way job trickling can work without stalling the pipeline: We would know how many jobs were pending for each @recombine job and which jobs go together.

## 2.11.7 Planned: Job Trickling brain storming Notes

- allows depth first iteration of tree
- @recombine is the necessary step, otherwise all @split + @merge / @collate end in a pipeline stall and we are back to running breadth first rather than depth first. Might as well not bother...
- Jobs need unique job\_id tag
- Need a way of generating filenames without returning from a function indefinitely: i.e. @originate and @split should yield
- Need a way of knowing which files group together (i.e. were split from a common job) without using regex (magic @split and @remerge)
- @split needs to be able to specify at run time the number of resulting jobs without using wild cards
- @merge needs to know when all of a group of files have completed
- legacy support for wild cards and file names.
- Possible breaking change: Assumes an explicit @follows if require *all* jobs from the previous task to finish
- “Push” system of checking in completed jobs into “slots” of waiting tasks
- New jobs dispatched when slots filled adequately
- Funny “single file” mode for @transform, @files needs to be regularised so it is a syntactic (front end) convenience (oddity!) and not plague the inards of ruffus

- use named parameters in decorators for clarity?

## 2.11.8 Planned: Custom parameter generator

Leverages built-in Ruffus functionality. Don't have to write entire parameter generation from scratch.

- Gets passed an iterator where you can do a for loop to get input parameters / a flattened list of files
- Other parameters are forwarded as is
- The duty of the function is to `yield` input, output, extra parameters

Simple to do but how do we prevent this from being a job-trickling barrier?

Postpone until we have an initial design for job-trickling: Ruffus v.4 ;-(

## 2.11.9 Desired!: Ruffus GUI interface.

Desktop (PyQT or web-based solution?) I'd love to see an svg pipeline picture that I could actually interact with

## 2.11.10 Find contributions for!: Extending graphviz output

### 2.11.11 Desired!: Deleting intermediate files

### 2.11.12 Desired!: Registering jobs for clean up

## 2.12 Implementation Tips

### 2.12.1 how to write new decorators

New placeholder class. E.g. for `@new_deco`

```
class new_deco(task_decorator):  
    pass
```

Add to list of action names and ids:

```
action_names = ["unspecified",  
                ...  
                "task_new_deco",  
  
                action_task_new_deco      = 15
```

Add function:

```
def task_transform (self, orig_args):
```

Add documentation to:

- `decorators/NEW_DECORATOR.rst`
- `decorators/decorators.rst`
- `_templates/layout.html`
- `manual`

## 2.13 Implementation notes

### 2.13.1 Refactoring: parameter handling

**Though the code is still split in a not very sensible way between `ruffus_utility.py`, `file_name_parameters.py`**  
 some rationalisation has taken place, and comments added so further refactoring can be made more easily.

Common code for:

```
file_name_parameters.split_ex_param_factory()
file_name_parameters.transform_param_factory()
file_name_parameters.collate_param_factory()
```

has been moved to `file_name_parameters.py.yield_io_params_per_job()`

unit tests added to `test_file_name_parameters.py` and `test_ruffus_utility.py`

### 2.13.2 formatter

`get_all_paths_components(paths, regex_str)` in `ruffus_utility.py`

Input files names are first squished into a flat list of files. `get_all_paths_components()` returns both the regular expression matches and the break down of the path.

In case of name clashes, the classes with higher priority override:

1. Captures by name
2. Captures by index
3. **Path components:** ‘ext’ = extension with dot ‘basename’ = file name without extension  
‘path’ = path before basename, not ending with slash ‘subdir’ = list of directories starting with the most nested and ending with the root (if normalised) ‘subpath’ = list of ‘path’ with successive directories removed starting with the most nested and ending with the root (if normalised)

E.g. `name = '/a/b/c/sample1.bam', formatter=r"(.*) (?P<id>\d+)\.(.+)"`  
 returns:

```
0:      '/a/b/c/sample1.bam',           // Entire match captured by index
1:      '/a/b/c/sample',               // captured by index
2:      'bam',                      // captured by index
'id':   '1',                        // captured by name
'ext':  '.bam',
'subdir': ['c', 'b', 'a', '/'],
'subpath': ['/a/b/c', '/a/b', '/a', '/'],
'path':  '/a/b/c',
'basename': 'sample1',
```

The code is in `ruffus_utility.py`:

```
results = get_all_paths_components(paths, regex_str)
string.format(results[2])
```

All the magic is hidden inside black boxes `filename_transform` classes:

```
class t_suffix_filename_transform(t_filename_transform):
class t_regex_filename_transform(t_filename_transform):
class t_format_filename_transform(t_filename_transform):
```

#### formatter(): regex() and suffix()

The previous behaviour with `regex()` where mismatches fail even if no substitution is made is retained by the use of `re.subn()`. This is a corner case but I didn't want user code to break

```
# filter on ".txt"
input_filenames = ["a.wrong", "b.txt"]
regex("(\\.txt) \$")
# fails, no substitution possible
r"\1"
# fails anyway even through regular expression matches not referenced...
r"output.filename"
```

### 2.13.3 @product()

- Use combinatoric generators from `itertools` and keep that naming scheme
- Put all new generators in an `combinatorics` submodule namespace to avoid breaking user code. (They can be imported if necessary.)
- test code in `test/test_combinatorics.py`
- The `itertools.product(repeat)` parameter doesn't make sense for Ruffus and will not be used
- Flexible number of pairs of `task / glob / file names + formatter()`
- Only `formatter([OPTIONAL_REGEX])` provides the necessary flexibility to construct the output so we won't bother with suffix and regex
- Similar to `@transform` but with extra level of nested-ness

**Retain same code for @product and @transform by adding an additional level of indirection:**

- generator wrap around `get_strings_in_nested_sequence` to convert nested input parameters either to a single flat list of file names or to nested lists of file names

```
file_name_parameters.input_param_to_file_name_list (input_params)
file_name_parameters.list_input_param_to_file_name_list (input_params)
```

- `t_file_names_transform` class which stores a list of regular expressions, one for each `formatter()` object corresponding to a single set of input parameters

```
t_formatter_file_names_transform
t_nested_formatter_file_names_transform
```

- string substitution functions which will apply a list of `formatter` changes

```
ruffus.utility.t_formatter_replace()
ruffus.utility.t_nested_formatter_replace()
```

- `ruffus.utility.swap_doubly_nested_order()` makes the syntax / implementation very orthogonal

## 2.13.4 @permutations(...), @combinations(...), @combinations\_with\_replacement()

Similar to `@product` extra level of nested-ness is self versus self

### Retain same code for `@product`

- forward to a sinble `file_name_parameters.combinatorics_param_factory()`
- use `combinatorics_type` to dispatch to `combinatorics.permutations`, `combinatorics.combinations` and `combinatorics.combinations_with_replacement`
- use `list_input_param_to_file_name_list` from `file_name_parameters.product_param_factory()`

## 2.13.5 drmaa alternatives

Alternative, non-drmaa polling code at

[https://github.com/bjpop/rubra/blob/master/rubra/cluster\\_job.py](https://github.com/bjpop/rubra/blob/master/rubra/cluster_job.py)

## 2.13.6 Task completion monitoring

### How easy is it to abstract out the database?

- The database is Jacob Sondergaard's `dbdict` which is a nosql / key-value store wrapper around sqlite

```
job_history = dbdict.open(RUFFUS_HISTORY_FILE, picklevalues=True)
```

- The key is the output file name, so it is important not to confuse Ruffus by having different tasks generate the same output file!
- Is it possible to abstract this so that `jobs` get timestamped as well?
- If we should ever want to abstract out `dbdict`, we need to have a similar key-value store class, and make sure that a single instance of `dbdict` is used through `pipeline_run` which is passed up and down the function call chain. `dbdict` would then be drop-in replaceable by our custom (e.g. flat-file-based) `dbdict` alternative.

To peek into the database:

```
$ sqlite3 .ruffus_history.sqlite
sqlite> .tables
data
sqlite> .schema data
CREATE TABLE data (key PRIMARY KEY,value);
sqlite> select key from data order by key;
```

### Can we query the database, get Job history / stats?

Yes, if we write a function to read and dump the entire database but this is only useful with timestamps and task names. See below

## What are the run time performance implications?

Should be fast: a single db connection is created and used inside pipeline\_run, pipeline\_printout, pipeline\_printout\_graph

## Avoid pauses between tasks

Allows Ruffus to avoid adding an extra 1 second pause between tasks to guard against file systems with low timestamp granularity.

- If the local file time looks to be in sync with the underlying file system, saved system time is used instead of file timestamps

## 2.13.7 @mkdir(...),

- mkdir continues to work seamlessly inside @follows) but also as its own decorator @mkdir due to the original happy orthogonal design
- fixed bug in checking so that Ruffus doesn't blow up if non strings are in the output (number...)
- note: adding the decorator to a previously undecorated function might have unintended consequences. The undecorated function turns into a zombie.
- fixed ugly bug in pipeline\_printout for printing single line output
- fixed description and printout indent

## 2.14 FAQ

### 2.14.1 Citations

#### Q. How should *Ruffus* be cited in academic publications?

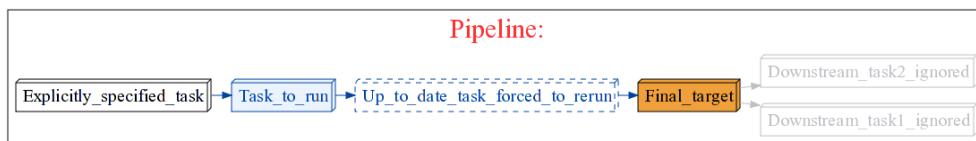
The official publication describing the original version of *Ruffus* is:

Leo Goodstadt (2010) : **Ruffus: a lightweight Python library for computational pipelines.**  
*Bioinformatics* 26(21): 2778-2779

### 2.14.2 General

#### Q. *Ruffus* won't create dependency graphs

A. You need to have installed dot from Graphviz to produce pretty flowcharts like this:



## Q. *Ruffus* seems to be hanging in the same place

A. If *ruffus* is interrupted, for example, by a Ctrl-C, you will often find the following lines of code highlighted:

```
File "build/bdist.linux-x86_64/egg/ruffus/task.py", line 1904, in pipeline_run
File "build/bdist.linux-x86_64/egg/ruffus/task.py", line 1380, in run_all_jobs_in_task
File "/xxxx/python2.6/multiprocessing/pool.py", line 507, in next
    self._cond.wait(timeout)
File "/xxxxx/python2.6/threading.py", line 237, in wait
    waiter.acquire()
```

This is *not* where *ruffus* is hanging but the boundary between the main programme process and the subprocesses which run *ruffus* jobs in parallel.

This is naturally where broken execution threads get washed up onto.

## Q. Regular expression substitutions don't work

A. If you are using the special regular expression forms "\1", "\2" etc. to refer to matching groups, remember to 'escape' the substitution pattern string. The best option is to use 'raw' python strings. For example:

```
r"\1_substitutes\2correctly\3four\4times"
```

Ruffus will throw an exception if it sees an unescaped "\1" or "\2" in a file name.

## Q. How to force a pipeline to appear up to date?

*I have made a trivial modification to one of my data files and now Ruffus wants to rerun my month long pipeline. How can I convince Ruffus that everything is fine and to leave things as they are?*

The standard way to do what you are trying to do is to touch all the files downstream... That way the modification times of your analysis files would postdate your existing files. You can do this manually but Ruffus also provides direct support:

```
pipeline_run (touch_files_only = True)
```

`pipeline_run` will run your script normally stepping over up-to-date tasks and starting with jobs which look out of date. However, after that, none of your pipeline task functions will be called, instead, each non-up-to-date file is `touch`-ed in turn so that the file modification dates follow on successively.

See the documentation for `pipeline_run()`

It is even simpler if you are using the new Ruffus.cmdline support from version 2.4. You can just type

```
your script --touch_files_only [--other_options_of_your_own_etc]
```

See *command line* documentation.

## Q. How can I use my own decorators with Ruffus?

(Thanks to Radhouane Aniba for contributing to this answer.)

1. With care! If the following two points are observed:

## 1. Use @wraps from functools or Michele Simionato's decorator module

These will automatically forward attributes from the task function correctly:

- `__name__` and `__module__` is used to identify functions uniquely in a Ruffus pipeline, and
- `pipeline_task` is used to hold per task data

## 2. Always write Ruffus decorators first before your own decorators.

Otherwise, your decorator will be ignored.

So this works:

```
@follows(prev_task)
@custom_decorator(something)
def test():
    pass
```

This is a bit futile

```
# ignore @custom_decorator
@custom_decorator(something)
@follows(prev_task)
def test():
    pass
```

This order dependency is an unfortunate quirk of how python decorators work. The last (rather futile) piece of code is equivalent to:

```
test = custom_decorator(something)(ruffus.follows(prev_task)(test))
```

Unfortunately, Ruffus has no idea that someone else (`custom_decorator`) is also modifying the `test()` function after it (`ruffus.follows`) has had its go.

### Example decorator:

Let us look at a decorator to time jobs:

```
import sys, time
def time_func_call(func, stream, *args, **kwargs):
    """prints elapsed time to standard out, or any other file-like object with a .write() method"""
    start = time.time()
    # Run the decorated function.
    ret = func(*args, **kwargs)
    # Stop the timer.
    end = time.time()
    elapsed = end - start
    stream.write("{} took {} seconds\n".format(func.__name__, elapsed))
    return ret

from ruffus import *
import sys
import time
```

```
@time_job(sys.stderr)
def first_task():
    print "First task"

@follows(first_task)
@time_job(sys.stderr)
def second_task():
    print "Second task"

@follows(second_task)
@time_job(sys.stderr)
def final_task():
    print "Final task"

pipeline_run()
```

What would @time\_job look like?

## 1. Using functools @wraps

```
import functools
def time_job(stream=sys.stdout):
    def actual_time_job(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            return time_func_call(func, stream, *args, **kwargs)
        return wrapper
    return actual_time_job
```

## 2. Using Michele Simionato's decorator module

```
import decorator
def time_job(stream=sys.stdout):
    def time_job(func, *args, **kwargs):
        return time_func_call(func, stream, *args, **kwargs)
    return decorator.decorator(time_job)
```

## 2. By hand, using a callable object

```
class time_job(object):
    def __init__(self, stream=sys.stdout):
        self.stream = stream
    def __call__(self, func):
        def inner(*args, **kwargs):
            return time_func_call(func, self.stream, *args, **kwargs)
        # remember to forward __name__
        inner.__name__ = func.__name__
        inner.__module__ = func.__module__
        inner.__doc__ = func.__doc__
        if hasattr(func, "pipeline_task"):
            inner.pipeline_task = func.pipeline_task
        return inner
```

```
    inner.pipeline_task = func.pipeline_task
    return inner
```

**Q. Can a task function in a *Ruffus* pipeline be called normally outside of Ruffus?**

A. Yes. Most python decorators wrap themselves around a function. However, *Ruffus* leaves the original function untouched and unwrapped. Instead, *Ruffus* adds a `pipeline_task` attribute to the task function to signal that this is a pipelined function.

This means the original task function can be called just like any other python function.

**Q. My *Ruffus* tasks create two files at a time. Why is the second one ignored in successive stages of my pipeline?**

*This is my code:*

```
from ruffus import *
import sys
@transform("start.input", regex("."), ("first_output.txt", "second_output.txt"))
def task1(i,o):
    pass

@transform(task1, suffix(".txt"), ".result")
def task2(i, o):
    pass

pipeline_printout(sys.stdout, [task2], verbose=3)
```

---

Tasks which will be run:

```
Task = task1
Job = [start.input
       ->[first_output.txt, second_output.txt]]

Task = task2
Job = [[first_output.txt, second_output.txt]
       ->first_output.result]
```

---

A: This code produces a single output of a tuple of 2 files. In fact, you want two outputs, each consisting of 1 file.

You want a single job (single input) to be produce multiple outputs (multiple jobs in downstream tasks). This is a one-to-many operation which calls for `@split`:

```
from ruffus import *
import sys
@split("start.input", ("first_output.txt", "second_output.txt"))
def task1(i,o):
    pass

@transform(task1, suffix(".txt"), ".result")
def task2(i, o):
    pass
```

```
pipeline_printout(sys.stdout, [task2], verbose=3)
```

---

```
Tasks which will be run:
```

```
Task = task1
    Job = [start.input
           ->[first_output.txt, second_output.txt]]

Task = task2
    Job = [first_output.txt
           ->first_output.result]
    Job = [second_output.txt
           ->second_output.result]
```

---

## Q. How can a *Ruffus* task produce output which goes off in different directions?

A. As above, anytime there is a situation which requires a one-to-many operation, you should reach for `@subdivide`. The advanced form takes a regular expression, making it easier to produce multiple derivatives of the input file. The following example subdivides 2 jobs each into 3, so that the subsequence task will run  $2 \times 3 = 6$  jobs.

```
from ruffus import *
import sys
@subdivide(["1.input_file",
            "2.input_file"],
            regex(r"(.).input_file"),      # match file prefix
            [r"\1.file_type1",
             r"\1.file_type2",
             r"\1.file_type3"])
def split_task(input, output):
    pass

@transform(split_task, regex("(.)"), r"\1.test")
def test_split_output(i, o):
    pass

pipeline_printout(sys.stdout, [test_split_output], verbose = 3)
```

Each of the original 2 files have been split in three so that `test_split_output` will run 6 jobs simultaneously.

---

```
Tasks which will be run:
```

```
Task = split_task
    Job = [1.input_file ->[1.file_type1, 1.file_type2, 1.file_type3]]
    Job = [2.input_file ->[2.file_type1, 2.file_type2, 2.file_type3]]

Task = test_split_output
    Job = [1.file_type1 ->1.file_type1.test]
    Job = [1.file_type2 ->1.file_type2.test]
    Job = [1.file_type3 ->1.file_type3.test]
    Job = [2.file_type1 ->2.file_type1.test]
```

```
Job = [2.file_type2 ->2.file_type2.test]
Job = [2.file_type3 ->2.file_type3.test]
```

---

## Q. Can I call extra code before each job?

- A. This is easily accomplished by hijacking the process for checking if jobs are up to date or not (@check\_if\_upToDate):

```
from ruffus import *
import sys

def run_this_before_each_job (*args):
    print "Calling function before each job using these args", args
    # Remember to delegate to the default *Ruffus* code for checking if
    #   jobs need to run.
    return needs_update_check_modify_time(*args)

@check_if_upToDate(run_this_before_each_job)
@files(([None, "a.1"], [None, "b.1"]))
def task_func(input, output):
    pass

pipeline_printout(sys.stdout, [task_func])
```

This results in:

---

```
>>> pipeline_run([task_func])
Calling function before each job using these args (None, 'a.1')
Calling function before each job using these args (None, 'a.1')
Calling function before each job using these args (None, 'b.1')
    Job = [None -> a.1] completed
    Job = [None -> b.1] completed
Completed Task = task_func
```

---

**Note:** Because `run_this_before_each_job(...)` is called whenever *Ruffus* checks to see if a job is up to date or not, the function may be called twice for some jobs (e.g. (None, 'a.1') above).

## Q. Does Ruffus allow checkpointing: to distinguish interrupted and completed results?

### A. Use the builtin sqlite checkpointing

By default, `pipeline_run(...)` will save the timestamps for output files from successfully run jobs to an sqlite database file (`.ruffus_history.sqlite`) in the current directory .

- If you are using `Ruffus.cmdline`, you can change the checksum / timestamp database file name on the command line using `--checksum_file_name NNNN`
- 

The level of timestamping / checksumming can be set via the `checksum_level` parameter:

```
pipeline_run(..., checksum_level = N, ...)
```

where the default is 1:

```
level 0 : Use only file timestamps  
level 1 : above, plus timestamp of successful job completion  
level 2 : above, plus a checksum of the pipeline function body  
level 3 : above, plus a checksum of the pipeline function default arguments and the additional arguments
```

## A. Use a flag file

When gmake is interrupted, it will delete the target file it is updating so that the target is remade from scratch when make is next run. Ruffus, by design, does not do this because, more often than not, the partial / incomplete file may be usefully if only to reveal, for example, what might have caused an interrupting error or exception. It also seems a bit too clever and underhand to go around the programmer's back to delete files...

A common *Ruffus* convention is to create an empty checkpoint or “flag” file whose sole purpose is to record a modification-time and the successful completion of a job.

This would be task with a completion flag:

The `flag_files` `xxx.stage2_finished` indicate that each job is finished. If this is missing, `xxx.stage2` is only a partial, interrupted result.

The only thing to be aware of is that the flag file will appear in the list of inputs of the downstream task, which should accordingly look like this:

```
cmd = ("do_something3 %(input_file)s >| %(task_output_file)s ")
cmd = cmd % {
    "input_file": input_file,
    "task_output_file": task_output_file
}
# completion flag file for this task
if not os.system( cmd ):
    open(flag_file, "w")
```

The *Bioinformatics example* contains code for checkpointing.

## A. Use a temp file

Thanks to Martin Goodson for suggesting this and providing an example. In his words:

“I normally use a decorator to create a temporary file which is only renamed after the task has completed without any problems. This seems a more elegant solution to the problem:”

```
def usetemp(task_func):
    """ Decorate a function to write to a tmp file and then rename it. So half finished task
    """
    @wraps(task_func)
    def wrapper_function(*args, **kwargs):
        args=list(args)
        outnames=args[1]
        if not isinstance(outnames, basestring) and hasattr(outnames, '__getitem__'):
            tmpnames=[str(x)+".tmp" for x in outnames]
            args[1]=tmpnames
        task_func(*args, **kwargs)
        try:
            for tmp, name in zip(tmpnames, outnames):
                if os.path.exists(tmp):
                    os.rename(tmp, str(name))
        except BaseException as e:
            for name in outnames:
                if os.path.exists(name):
                    os.remove(name)
            raise (e)
        else:
            tmp=str(outnames)+'.tmp'
            args[1]=tmp
            task_func(*args, **kwargs)
            os.rename(tmp, str(outnames))
    return wrapper_function
```

Use like this:

```
@files(None, 'client1.price')
@usetemp
def getusers(inputfile, outputname):
    ****
    # code goes here
    # outputname now refers to temporary file
    pass
```

### 2.14.3 Windows

#### Q. Windows seems to spawn *ruffus* processes recursively

A. It is necessary to protect the “entry point” of the program under windows. Otherwise, a new process will be started each time the main module is imported by a new Python interpreter as an unintended side effects. Causing a cascade of new processes.

See: <http://docs.python.org/library/multiprocessing.html#multiprocessing-programming>

This code works:

```
if __name__ == '__main__':
    try:
        pipeline_run([parallel_task], multiprocess = 5)
    except Exception, e:
        print e.args
```

### 2.14.4 Sun Grid Engine / PBS / SLURM etc

#### Q. Can Ruffus be used to manage a cluster or grid based pipeline?

1. Some minimum modifications have to be made to your *Ruffus* script to allow it to submit jobs to a cluster

See *ruffus.drmaa\_wrapper*

Thanks to Andreas Heger and others at CGAT and Bernie Pope for contributing ideas and code.

#### Q. When I submit lots of jobs via Sun Grid Engine (SGE), the head node occassionally freezes and dies

1. You need to use multithreading rather than multiprocessing. See *ruffus.drmaa\_wrapper*

#### Q. Keeping Large intermediate files

Sometimes pipelines create a large number of intermediate files which might not be needed later.

Unfortunately, the current design of *Ruffus* requires these files to hang around otherwise the pipeline will not know that it ran successfully.

We have some tentative plans to get around this but in the meantime, Bernie Pope suggests truncating intermediate files in place, preserving timestamps:

```
# truncate a file to zero bytes, and preserve its original modification time
def zeroFile(file):
    if os.path.exists(file):
        # save the current time of the file
        timeInfo = os.stat(file)
        try:
            f = open(file,'w')
        except IOError:
            pass
        else:
            f.truncate(0)
            f.close()
```

```
# change the time of the file back to what it was
os.utime(file, (timeInfo.st_atime, timeInfo.st_mtime))
```

## 2.14.5 Sharing python objects between Ruffus processes running concurrently

The design of Ruffus envisages that much of the data flow in pipelines occurs in files but it is also possible to pass python objects in memory.

Ruffus uses the `multiprocessing` module and much of the following is a summary of what is covered in depth in the Python Standard Library Documentation.

Running Ruffus using `pipeline_run(..., multiprocess = NNN)` where `NNN > 1` runs each job concurrently on up to `NNN` separate local processes. Each task function runs independently in a different python interpreter, possibly on a different CPU, in the most efficient way. However, this does mean we have to pay some attention to how data is sent across process boundaries (unlike the situation with `pipeline_run(..., multithread = NNN)` ).

The python code and data which comprises your multitasking Ruffus job is sent to a separate process in three ways:

1. The python function code and data objects are `pickled`, i.e. converting into a byte stream, by the master process, sent to the remote process before being converted back into normal python (`unpickling`).
2. The parameters for your jobs, i.e. what Ruffus calls your task functions with, are separately `pickled` and sent to the remote process via `multiprocessing.Queue`
3. You can share and synchronise other data yourselves. The canonical example is the logger provided by `Ruffus.cmdline.setup_logging`

---

**Note:** Check that your function code and data can be `pickled`.

Only functions, built-in functions and classes defined at the top level of a module are picklable.

---

The following answers are a short “how-to” for sharing and synchronising data yourselves.

### Can ordinary python objects be shared between processes?

1. Objects which can be `pickled` can be shared as is. These include
  - numbers
  - strings
  - tuples, lists, sets, and dictionaries containing only objects which can be `pickled`.
2. If these do not change during your pipeline, you can just use them without any further effort in your task.
3. If you need to use the value at the point when the task function is *called*, then you need to pass the python object as parameters to your task. For example:

```
# changing_list changes...
@transform(previous_task, suffix(".foo"), ".bar", changing_list)
def next_task(input_file, output_file, changing_list):
    pass
```

4. If you need to use the value when the task function is *run* then see *the following answer..*

## Why am I getting PicklingError?

What is happening? Didn't Joan of Arc solve this once and for all?

1. Some of the data or code in your function cannot be `pickled` and is being asked to be sent by python multprocessing across process boundaries.

When you run your pipeline using multiprocess:

```
pipeline_run([], verbose = 5, multiprocess = 5, logger = ruffusLoggerProxy)
```

You will get the following errors:

```
Exception in thread Thread-2:  
Traceback (most recent call last):  
  File "/path/to/python/python2.7/threading.py", line 808, in __bootstrap_inner  
    self.run()  
  File "/path/to/python/python2.7/threading.py", line 761, in run  
    self._target(*self._args, **self._kwargs)  
  File "/path/to/python/python2.7/multiprocessing/pool.py", line 342, in _handle_task  
    put(task)  
PicklingError: Can't pickle <type 'function'>: attribute lookup __builtin__.function
```

which go away when you set `pipeline_run([], multiprocess = 1, ...)`

Unfortunately, pickling errors are particularly ill-served by standard python error messages. The only really good advice is to take the offending code and try and `pickle` it yourself and narrow down the errors. Check your objects against the list in the `pickle` module. Watch out especially for nested functions. These will have to be moved to file scope. Other objects may have to be passed in proxy (see below).

## How about synchronising python objects in real time?

1. You can use managers and proxy objects from the `multiprocessing` module.

The underlying python object would be owned and managed by a (hidden) server process. Other processes can access the shared objects transparently by using proxies. This is how the logger provided by `Ruffus.cmdline.setup_logging` works:

```
# optional logger which can be passed to ruffus tasks  
logger, logger_mutex = cmdline.setup_logging (__name__, options.log_file, options.verbose)
```

`logger` is a proxy for the underlying python `logger` object, and it can be shared freely between processes.

The best course is to pass `logger` as a parameter to a *Ruffus* task.

The only caveat is that we should make sure multiple jobs are not writing to the log at the same time. To synchronise logging, we use proxy to a non-reentrant `multiprocessing.Lock`.

```
logger, logger_mutex = cmdline.setup_logging (__name__, options.log_file, options.verbose)

@transform(previous_task, suffix(".foo"), ".bar", logger, logger_mutex)
def next_task(input_file, output_file, logger, logger_mutex):
    with logger_mutex:
        logger.info("We are in the middle of next_task: %s -> %s" % (input_file, output_file))
```

## Can I share and synchronise my own python classes via proxies?

1. `multiprocessing.managers.SyncManager` provides out of the box support for lists, arrays and dicts etc.

Most of the time, we can use a “vanilla” manager provided by `multiprocessing.Manager()`:

```
import multiprocessing
manager = multiprocessing.Manager()

list_proxy = manager.list()
dict_proxy = manager.dict()
lock_proxy = manager.Lock()
namespace_proxy = manager.Namespace()
queue_proxy = manager.Queue([maxsize])
reentrant_lock_proxy = manager.RLock()
semaphore_proxy = manager.Semaphore([value])
char_array_proxy = manager.Array('c')
integer_proxy = manager.Value('i', 6)

@transform(previous_task, suffix(".foo"), ".bar", lock_proxy, dict_proxy, list_proxy)
def next_task(input_file, output_file, lock_proxy, dict_proxy, list_proxy):
    with lock_proxy:
        list_proxy.append(3)
        dict_proxy['a'] = 5
```

However, you can also create proxy custom classes for your own objects.

In this case you may need to derive from `multiprocessing.managers.SyncManager` and register proxy functions. See `Ruffus.proxy_logger` for an example of how to do this.

## How do I send python objects back and forth without tangling myself in horrible synchronisation code?

1. Sharing python objects by passing messages is a much more modern and safer way to coordinate multitasking than using synchronization primitives like locks.

The python `multiprocessing` module provides support for passing python objects as messages between processes. You can either use `pipes` or `queues`. The idea is that one process pushes an object onto a `pipe` or `queue` and the other processes pops it out at the other end. `Pipes` are only two ended so `queues` are usually a better fit for sending data to multiple Ruffus jobs.

Proxies for `queues` can be passed between processes as in the previous section

## How do I share large amounts of data efficiently across processes?

1. If it is really impractical to use data files on disk, you can put the data in shared memory.

It is possible to create shared objects using shared memory which can be inherited by child processes or passed as Ruffus parameters. This is probably most efficiently done via the `array` interface. Again, it is easy to create locks and proxies for synchronised access:

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

manager = multiprocessing.Manager()
```

```
lock_proxy      = manager.Lock()
int_array_proxy = manager.Array('i', [123] * 100)

@transform(previous_task, suffix(".foo"), ".bar", lock_proxy, int_array_proxy)
def next_task(input_file, output_file, lock_proxy, int_array_proxy):
    with lock_proxy:
        int_array_proxy[23] = 71
```

## 2.15 Glossary

**task** A stage in a computational pipeline.

Each **task** in *ruffus* is represented by a python function.

For example, a task might be to find the products of a sets of two numbers:

```
4 x 5 = 20
5 x 6 = 30
2 x 7 = 14
```

**job** Any number of operations which can be run in parallel and make up the work in a stage of a computational pipeline.

Each **task** in *ruffus* is a separate call to the **task** function.

For example, if a task is to find products of numbers, each of these will be a separate job.

Job1:

```
4 x 5 = 20
```

Job2:

```
5 x 6 = 30
```

Job3:

```
2 x 7 = 14
```

Jobs need not complete in order.

**decorator** Ruffus decorators allow functions to be incorporated into a computational pipeline, with automatic generation of parameters, dependency checking etc., without modifying any code within the function. Quoting from the [python wiki](#):

A Python decorator is a specific change to the Python syntax that allows us to more conveniently alter functions and methods.

Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated.

**generator** python generators are new to python 2.2 (see [Charming Python: Iterators and simple generators](#)). They allow iterable data to be generated on the fly.

Ruffus asks for generators when you want to generate **job** parameters dynamically.

Each set of job parameters is returned by the `yield` keyword for greater clarity. For example,:

```
def generate_job_parameters():

    for file_index, file_name in iterate(all_file_names):
```

```
# parameter for each job
yield file_index, file_name
```

Each job takes the parameters file\_index and file\_name



# Ruffus

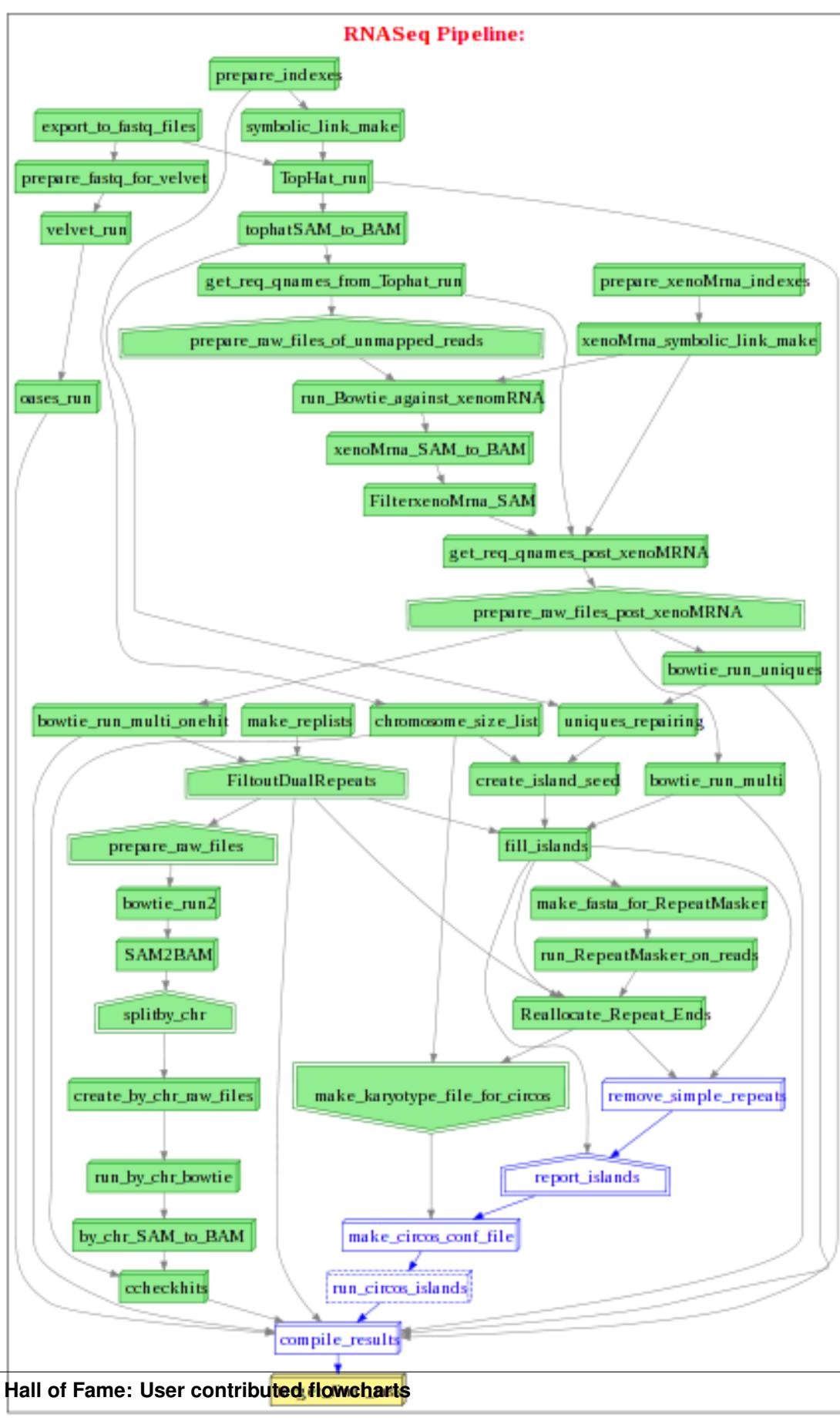
## 2.16 Hall of Fame: User contributed flowcharts

Please contribute your own work flows in your favourite colours with (an optional) short description to email: ruffus\_lib at llew.org.uk

### 2.16.1 RNASeq pipeline

<http://en.wikipedia.org/wiki/RNA-Seq>

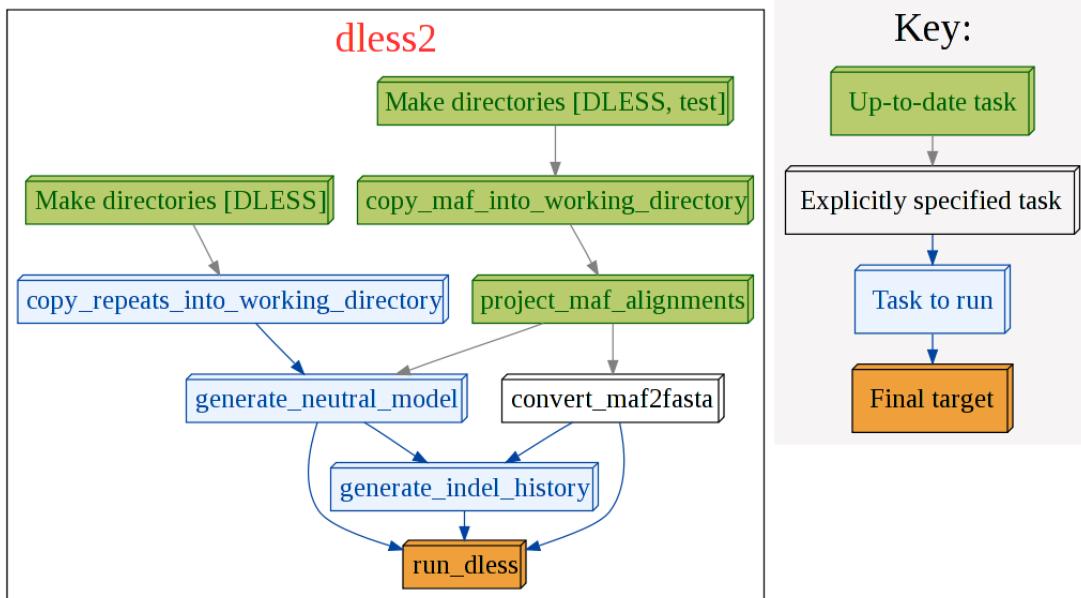
Mapping transcripts onto genomes using high-throughput sequencing technologies (svg).



## 2.16.2 non-coding evolutionary constraints

[http://en.wikipedia.org/wiki/Noncoding\\_DNA](http://en.wikipedia.org/wiki/Noncoding_DNA)

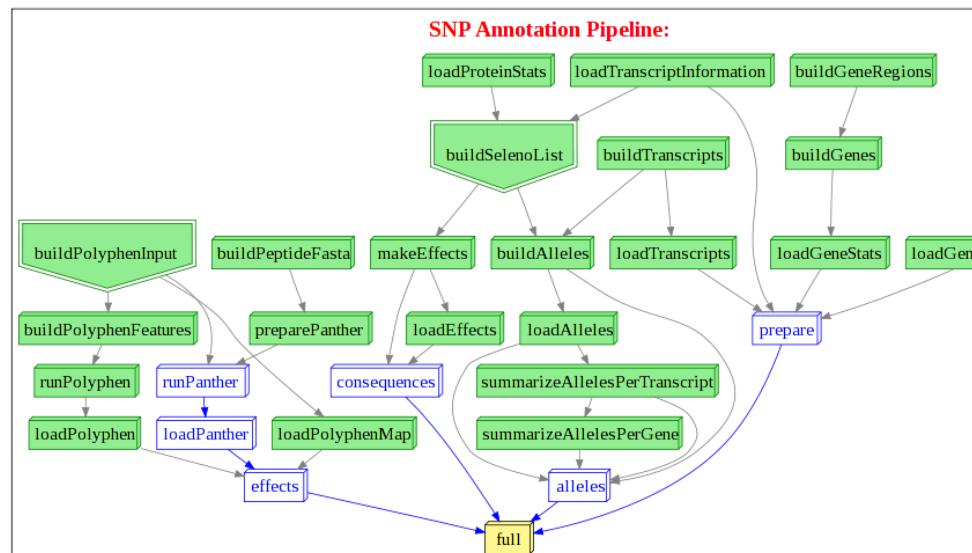
Non-protein coding evolutionary constraints in different species (svg).



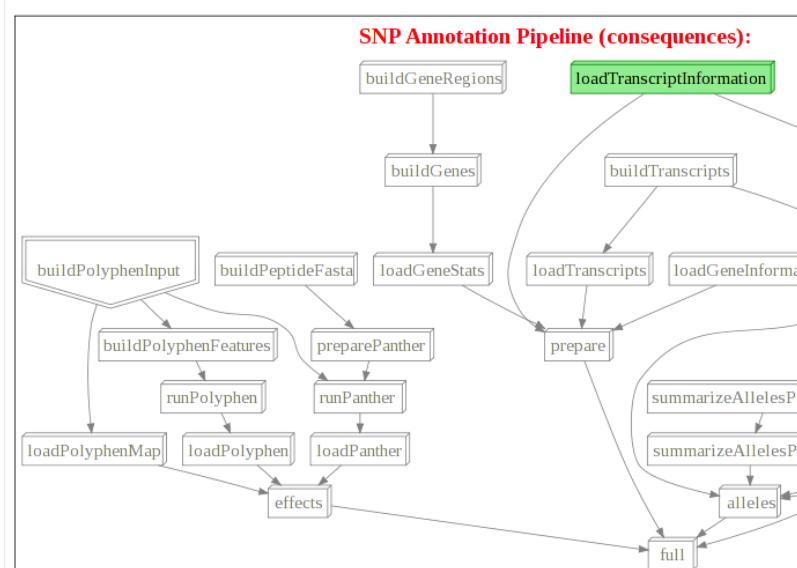
## 2.16.3 SNP annotation

Predicting impact of different Single Nucleotide Polymorphisms

[http://en.wikipedia.org/wiki/Single-nucleotide\\_polymerism](http://en.wikipedia.org/wiki/Single-nucleotide_polymerism)



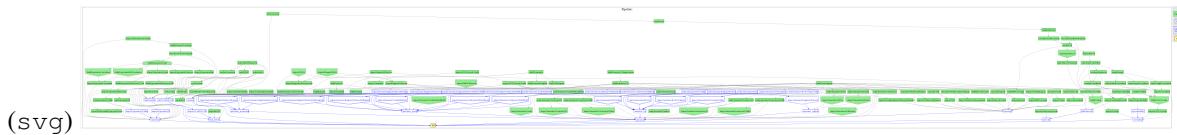
Population variation across genomes (svg).



Using “pseudo” targets to run only part of the pipeline (svg).

## 2.16.4 Chip-Seq analysis

Analysing DNA binding sites with Chip-Seq <http://en.wikipedia.org/wiki/Chip-Sequencing>



## 2.17 Why *Ruffus*?

**Cylindrophis ruffus** is the name of the red-tailed pipe snake (bad python-y pun) which can be found in Hong Kong where the original author comes from.

*Ruffus* is a shy creature, and pretends to be a cobra or a banded krait by putting up its red tail and ducking its head in its coils when startled.

	
<ul style="list-style-type: none"><li>• Not venomous</li><li>• <b>Mostly Harmless</b></li></ul>	<ul style="list-style-type: none"><li>• Deadly poisonous</li><li>• <b>Seriously unfriendly</b></li></ul>

Be careful not to step on one when running down country park lanes at full speed in Hong Kong: this snake is a [rare breed!](#)

*Ruffus* does most of its work at night and sleeps during the day: typical of many (but alas not all) python programmers!

The original [red-tail pipe](#) and [banded krait](#) images are from wikimedia.

## EXAMPLES

### 3.1 Construction of a simple pipeline to run BLAST jobs

#### 3.1.1 Overview

This is a simple example to illustrate the convenience **Ruffus** brings to simple tasks in bioinformatics.

1. **Split** a problem into multiple fragments that can be
2. **Run in parallel** giving partial solutions that can be
3. **Recombined** into the complete solution.

The example code runs a `ncbi blast` search for four sequences against the human `refseq` protein sequence database.

1. **Split** each of the four sequences into a separate file.
2. **Run in parallel** Blastall on each sequence file
3. **Recombine** the BLAST results by simple concatenation.

In real life,

- **BLAST** already provides support for multiprocessing
- Sequence files would be split in much larger chunks, with many sequences
- The jobs would be submitted to large computational farms (in our case, using the SunGrid Engine).
- The High Scoring Pairs (HSPs) would be parsed / filtered / stored in your own formats.

---

**Note:** This bioinformatics example is intended to showcase *some* of the features of Ruffus.

1. For a more gentle introduction, please consult the *simple tutorial*.
  2. The *manual* covers all the features in Ruffus.
- 

#### 3.1.2 Prerequisites

##### 1. Ruffus

To install Ruffus on most systems with python installed:

```
easy_install -U ruffus
```

Otherwise, [download](#) Ruffus and run:

```
tar -xvzf ruffus-xxx.tar.gz
cd ruffus-xxx
./setup install
```

where xxx is the latest Ruffus version.

## 2. BLAST

This example assumes that the `BLAST` `blastall` and `formatdb` executables are installed and on the search path. Otherwise download from [here](#).

## 3. human refseq sequence database

We also need to download the human refseq sequence file and format the ncbi database:

```
wget ftp://ftp.ncbi.nih.gov/refseq/H_sapiens/mRNA_Prot/human.protein.faa.gz
gunzip human.protein.faa.gz

formatdb -i human.protein.faa
```

## 4. test sequences

Query sequences in FASTA format can be found in `original.fa`

### 3.1.3 Code

The code for this example can be found [here](#) and pasted into the python command shell.

### 3.1.4 Step 1. Splitting up the query sequences

We want each of our sequences in the query file `original.fa` to be placed in a separate files named `XXX.segment` where `XXX` = 1 -> the number of sequences.

```
current_file_index = 0
for line in open("original.fa"):
    # start a new file for each accession line
    if line[0] == '>':
        current_file_index += 1
        current_file = open("%d.segment" % current_file_index, "w")
        current_file.write(line)
```

To use this in a pipeline, we only need to wrap this in a function, “decorated” with the Ruffus keyword `@split`:

**@split** *I-to-many* operation: Each “original.fa” is split into many “\*.segment”

```
@split("original.fa", "*.segment")
def splitFasta (seqFile, segments):

    current_file_index = 0
    for line in open(original_fasta):
        #
        # start a new file for each accession line
        #
        if line[0] == '>':
            current_file_index += 1
            current_file = open("%d.segment" % current_file_index, "w")
            current_file.write(line)
```

This indicates that we are splitting up the input file original.fa into however many \*.segment files as it takes.

The pipelined function itself takes two arguments, for the input and output.

We shall see later this simple `@split` decorator already gives all the benefits of:

- Dependency checking
- Flowchart printing

### 3.1.5 Step 2. Run BLAST jobs in parallel

Assuming that blast is already installed, sequence matches can be found with this python code:

```
os.system("blastall -p blastp -d human.protein.faa -i 1.segment > 1.blastResult")
```

To pipeline this, we need to simply wrap in a function, decorated with the **Ruffus** keyword `@transform`.  
**@transform** *I-to-I* operation:

Each file from `splitFasta` with a suffix of “.segment” is transformed to a file with the suffix “.blastResult”

```
@transform(splitFasta, suffix(".segment"), ".blastResult")
def runBlast(seqFile, blastResultFile):

    os.system("blastall -p blastp -d human.protein.faa "+
              "-i %s > %s" % (seqFile, blastResultFile))
```

This indicates that we are taking all the output files from the previous `splitFasta` operation (\*.segment) and `@transform`-ing each to a new file with the .blastResult suffix. Each of these transformation operations can run in parallel if specified.

### 3.1.6 Step 3. Combining BLAST results

The following python code will concatenate the results together

```
output_file = open("final.blast_results", "w")
for i in glob("*.blastResults"):
    output_file.write(open(i).read())
```

To pipeline this, we need again to decorate with the **Ruffus** keyword `@merge`.  
`@merge` many-to-1 operation:

All files from `runBlast` are combined to give “`final.blast_results`”

```
@merge(runBlast, "final.blast_results")
def combineBlastResults (blastResultFiles, combinedBlastResultFile):

    output_file = open(combinedBlastResultFile, "w")
    for i in blastResultFiles:
        output_file.write(open(i).read())
```

This indicates that we are taking all the output files from the previous `runBlast` operation (`*.blastResults`) and `@merge`-ing them to the new file `final.blast_results`.

### 3.1.7 Step 4. Running the pipeline

We can run the completed pipeline using a maximum of 4 parallel processes by calling `pipeline_run`:

```
pipeline_run([combineBlastResults], verbose = 2, multiprocess = 4)
```

Though we have only asked Ruffus to run `combineBlastResults`, it traces all the dependencies of this task and runs all the necessary parts of the pipeline.

---

**Note:** The full code for this example can be found [here](#) suitable for pasting into the python command shell.

---

The `verbose` parameter causes the following output to be printed to stderr as the pipeline runs:

```
>>> pipeline_run([combineBlastResults], verbose = 2, multiprocess = 4)
Job = [original.fa -> *.segment] completed
Completed Task = splitFasta
Job = [1.segment -> 1.blastResult] completed
Job = [3.segment -> 3.blastResult] completed
Job = [2.segment -> 2.blastResult] completed
Job = [4.segment -> 4.blastResult] completed
Completed Task = runBlast
Job = [[1.blastResult, 2.blastResult, 3.blastResult, 4.blastResult] -> final.blast_results]
Completed Task = combineBlastResults
```

### 3.1.8 Step 5. Testing dependencies

If we invoked `pipeline_run` again, nothing further would happen because the pipeline is now up-to-date. But what if the pipeline had not run to completion?

We can simulate the failure of one of the `blastall` jobs by deleting its results:

```
os.unlink("4.blastResult")
```

Let us use the `pipeline_printout` function to print out the dependencies of the pipeline at a high `verbose` level which will show both complete and incomplete jobs:

```
>>> import sys
>>> pipeline_printout(sys.stdout, [combineBlastResults], verbose = 4)

Tasks which are up-to-date:

Task = splitFasta
    "Split sequence file into as many fragments as appropriate depending on the size of
        original_fasta"

Tasks which will be run:

Task = runBlast
    "Run blast"
        Job = [4.segment
            ->4.blastResult]
        Job needs update: Missing file 4.blastResult

Task = combineBlastResults
    "Combine blast results"
        Job = [[1.blastResult, 2.blastResult, 3.blastResult, 4.blastResult]
            ->final.blast_results]
        Job needs update: Missing file 4.blastResult
```

---

Only the parts of the pipeline which involve the missing BLAST result will be rerun. We can confirm this by invoking the pipeline.

```
>>> pipeline_run([combineBlastResults], verbose = 2, multiprocess = 4)

Job = [1.segment -> 1.blastResult] unnecessary: already up to date
Job = [2.segment -> 2.blastResult] unnecessary: already up to date
Job = [3.segment -> 3.blastResult] unnecessary: already up to date
Job = [4.segment -> 4.blastResult] completed
Completed Task = runBlast
    Job = [[1.blastResult, 2.blastResult, 3.blastResult, 4.blastResult] -> final.blast_results]
Completed Task = combineBlastResults
```

### 3.1.9 What is next?

In the *next (short) part*, we shall add some standard (boilerplate) code to turn this BLAST pipeline into a (slightly more) useful python program.

## 3.2 Part 2: A slightly more practical pipeline to run blasts jobs

### 3.2.1 Overview

Previously, we had built a simple pipeline to split up a FASTA file of query sequences so that these can be matched against a sequence database in parallel.

We shall wrap this code so that

- It is more robust to interruptions
- We can specify the file names on the command line

### 3.2.2 Step 1. Cleaning up any leftover junk from previous pipeline runs

We split up each of our sequences in the query file original.fa into a separate files named XXX.segment where XXX is the number of sequences in the FASTA file.

However, if we start with 6 sequences (giving 1.segment ... 6.segment), and we then edited original.fa so that only 5 were left, the file 6.segment would still be left hanging around as an unwanted, extraneous and confusing orphan.

As a general rule, it is a good idea to clean up the results of a previous run in a @split operation:

```
@split("original.fa", "*.segment")
def splitFasta (seqFile, segments):

    #
    #   Clean up any segment files from previous runs before creating new one
    #
    for i in glob.glob("*.segment"):
        os.unlink(i)

    # code as before...
```

### 3.2.3 Step 2. Adding a “flag” file to mark successful completion

When pipelined tasks are interrupted half way through an operation, the output may only contain part of the results in an incomplete or inconsistent state. There are three general options to deal with this:

1. Catch any interrupting conditions and delete the incomplete output
2. Tag successfully completed output with a special marker at the end of the file
3. Create an empty “flag” file whose only point is to signal success

Option (3) is the most reliable way and involves the least amount of work in Ruffus. We add flag files with the suffix .blastSuccess for our parallel BLAST jobs:

```
@transform(splitFasta, suffix(".segment"), [".blastResult", ".blastSuccess"])
def runBlast(seqFile, output_files):

    blastResultFile, flag_file = output_files

    #
    #   Existing code unchanged
    #
    os.system("blastall -p blastp -d human.protein.faa "+
              "-i %s > %s" % (seqFile, blastResultFile))

    #
    #   "touch" flag file to indicate success
    #
    open(flag_file, "w")
```

### 3.2.4 Step 3. Allowing the script to be invoked on the command line

We allow the query sequence file, as well as the sequence database and end results to be specified at runtime using the standard python `optparse` module. We find this approach to run time arguments generally useful for many Ruffus scripts. The full code can be *viewed here* and downloaded from `run_parallel_blast.py`.

The different options can be inspected by running the script with the `--help` or `-h` argument.

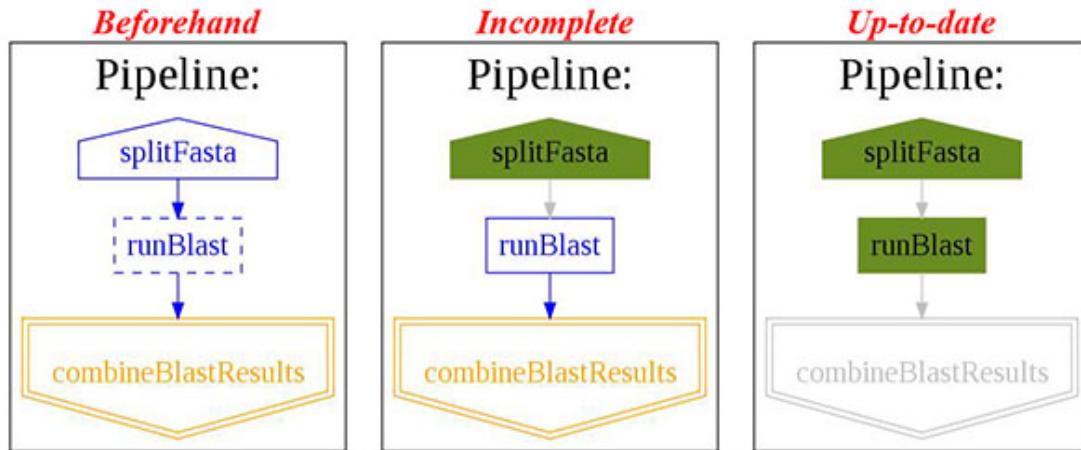
The following options are useful for developing Ruffus scripts:

```
--verbose | -v      : Print more detailed messages for each additional verbose level.  
E.g. run_parallel_blast --verbose --verbose --verbose ... (or -vvv)  
  
--jobs | -j        : Specifies the number of jobs (operations) to run in parallel.  
  
--flowchart FILE   : Print flowchart of the pipeline to FILE. Flowchart format  
depends on extension. Alternatives include ("dot", ".jpg",  
"*.svg", "*.png" etc). Formats other than "dot" require  
the dot program to be installed (http://www.graphviz.org/).  
  
--just_print | -n  Only print a trace (description) of the pipeline.  
The level of detail is set by --verbose.
```

### 3.2.5 Step 4. Printing out a flowchart for the pipeline

The `--flowchart` argument results in a call to `pipeline_printout_graph(...)`. This prints out a flowchart of the pipeline. Valid formats include `".dot"`, `".jpg"`, `".svg"`, `".png"` but all except for the first require the `dot` program to be installed (<http://www.graphviz.org/>).

The state of the pipeline is reflected in the flowchart:



### 3.2.6 Step 5. Errors

Because Ruffus scripts are just normal python functions, you can debug them using your usual tools, or jump to the offending line(s) even when the pipeline is running in parallel.

For example, these are the what the error messages would look like if we had mis-spelt `blastal`. In `run_parallel_blast.py`, python exceptions are raised if the `blastall` command fails.

Each of the exceptions for the parallel operations are printed out with the offending lines (line 204), and problems (blastal not found) highlighted in red.

```
Traceback (most recent call last):
  File "./run_parallel_blast.py", line 256, in <module>
    logger = logger, verbose=options.verbose)
  File "build/bdist.linux-i686/egg/ruffus/task.py", line 2655, in pipeline_run
ruffus.ruffus_exceptions.RethrownJobError:

Exceptions running jobs for

'def runBlast(...):'

Original exceptions:
Exception #1
exceptions.Exception(Failed to run 'blastal -p blastp -d human.protein.faa
                     -i tmp/1.segment > tmp/1.blastResult'
/bin/sh: blastal: not found
for __main__.runBlast.Job = [tmp/1.segment -> [tmp/1.blastResult,tmp/1.blastSuccess]]

Traceback (most recent call last):
[...]
  File "./run_parallel_blast.py", line 204, in runBlast
    run_cmd("blastal -p blastp -d human.protein.faa -i %s > %s" % (seqFile,
                     blastResultFile))
[...]

Exception #2
exceptions.Exception(Failed to run 'blastal -p blastp -d human.protein.faa
                     -i tmp/2.segment > tmp/2.blastResult'
/bin/sh: blastal: not found
for __main__.runBlast.Job = [tmp/2.segment -> [tmp/2.blastResult, tmp/2.blastSuccess]]

Traceback (most recent call last):
[...]
  File "./run_parallel_blast.py", line 204, in runBlast
    run_cmd("blastal -p blastp -d human.protein.faa -i %s > %s" % (seqFile,
                     blastResultFile))
[...]

Exception #3
exceptions.Exception(Failed to run 'blastal -p blastp -d human.protein.faa
                     -i tmp/3.segment > tmp/3.blastResult'
/bin/sh: blastal: not found
for __main__.runBlast.Job = [tmp/3.segment -> [tmp/3.blastResult, tmp/3.blastSuccess]]

Traceback (most recent call last):
[...]
  File "./run_parallel_blast.py", line 204, in runBlast
    run_cmd("blastal -p blastp -d human.protein.faa -i %s > %s" % (seqFile,
                     blastResultFile))
[...]
```

### 3.2.7 Step 6. Will it run?

The full code can be *viewed here* and downloaded from run\_parallel\_blast.py.

### 3.3 Ruffus code

```

import os, sys

exe_path = os.path.split(os.path.abspath(sys.argv[0]))[0]
sys.path.insert(0, os.path.abspath(os.path.join(exe_path, "...", "...", "...")))

from ruffus import *

original_fasta = "original.fa"
database_file = "human.protein.faa"

@split(original_fasta, "*.{segment}")
def splitFasta (seqFile, segments):
    """Split sequence file into
       as many fragments as appropriate
       depending on the size of original_fasta"""
    current_file_index = 0
    for line in open(original_fasta):
        #
        # start a new file for each accession line
        #
        if line[0] == '>':
            current_file_index += 1
            current_file = open("%d.segment" % current_file_index, "w")
        current_file.write(line)

@transform(splitFasta, suffix(".segment"), ".blastResult")
def runBlast(seqFile, blastResultFile):
    """Run blast"""
    os.system("blastall -p blastp -d %s -i %s > %s" %
              (database_file, seqFile, blastResultFile))

@merge(runBlast, "final.blast_results")
def combineBlastResults (blastResultFiles, combinedBlastResultFile):
    """Combine blast results"""
    output_file = open(combinedBlastResultFile, "w")
    for i in blastResultFiles:
        output_file.write(open(i).read())

pipeline_run([combineBlastResults], verbose = 2, multiprocess = 4)

#
#   Simulate interuption of the pipeline by
#       deleting the output of one of the BLAST jobs
#
os.unlink("4.blastResult")

pipeline_printout(sys.stdout, [combineBlastResults], verbose = 4)

#

```

```
#     Re-running the pipeline
#
# pipeline_run([combineBlastResults], verbose = 2, multiprocess = 4)
```

## 3.4 Ruffus code

```

parser = OptionParser(version="%prog 1.0", usage = "%prog --input_file QUERY_FASTA --database_file database_file --result_file result_file --temp_directory temp_directory -v -vv -vvv -n -j --flowchart --just_print -d -i")
parser.add_option("-i", "--input_file", dest="input_file",
    metavar="FILE",
    type="string",
    help="Name and path of query sequence file in FASTA format. ")
parser.add_option("-d", "--database_file", dest="database_file",
    metavar="FILE",
    type="string",
    help="Name and path of FASTA database to search. ")
parser.add_option("--result_file", dest="result_file",
    metavar="FILE",
    type="string",
    default="final.blast_results",
    help="Name and path of where the files should end up. ")
parser.add_option("-t", "--temp_directory", dest="temp_directory",
    metavar="PATH",
    type="string",
    default="tmp",
    help="Name and path of temporary directory where calculations "
         "should take place. ")

#
#   general options: verbosity / logging
#
parser.add_option("-v", "--verbose", dest = "verbose",
    action="count", default=0,
    help="Print more detailed messages for each additional verbose level."
         " E.g. run_parallel_blast --verbose --verbose --verbose ... (or -vvv)")

#
#   pipeline
#
parser.add_option("-j", "--jobs", dest="jobs",
    default=1,
    metavar="jobs",
    type="int",
    help="Specifies the number of jobs (operations) to run in parallel.")
parser.add_option("--flowchart", dest="flowchart",
    metavar="FILE",
    type="string",
    help="Print flowchart of the pipeline to FILE. Flowchart format "
         "depends on extension. Alternatives include ('.dot', '.jpg', "
         "'*.svg', '*.png' etc). Formats other than '.dot' require "
         "the dot program to be installed (http://www.graphviz.org/).")
parser.add_option("-n", "--just_print", dest="just_print",
    action="store_true", default=False,
    help="Only print a trace (description) of the pipeline. "
         "The level of detail is set by --verbose.")

(options, remaining_args) = parser.parse_args()

if not options.flowchart:
    if not options.database_file:
        parser.error("\n\n\tMissing parameter --database_file FILE\n\n")
    if not options.input_file:
        parser.error("\n\n\tMissing parameter --input_file FILE\n\n")

```

```

# imports

from ruffus import *
import subprocess

# Functions

def run_cmd(cmd_str):
    """
    Throw exception if run command fails
    """
    process = subprocess.Popen(cmd_str, stdout = subprocess.PIPE,
                               stderr = subprocess.PIPE, shell = True)
    stdout_str, stderr_str = process.communicate()
    if process.returncode != 0:
        raise Exception("Failed to run '%s'\n%sNon-zero exit status %s" %
                        (cmd_str, stdout_str, stderr_str, process.returncode))

# Logger

import logging
logger = logging.getLogger("run_parallel_blast")
#
# We are interesting in all messages
#
if options.verbose:
    logger.setLevel(logging.DEBUG)
    stderrhandler = logging.StreamHandler(sys.stderr)
    stderrhandler.setFormatter(logging.Formatter("% (message)s"))
    stderrhandler.setLevel(logging.DEBUG)
    logger.addHandler(stderrhandler)

# Pipeline tasks

original_fasta = options.input_file

```

```

database_file = options.database_file
temp_directory = options.temp_directory
result_file = options.result_file

@follows(mkdir(temp_directory))

@splits(original_fasta, os.path.join(temp_directory, "*.segment"))
def splitFasta (seqFile, segments):
    """Split sequence file into
       as many fragments as appropriate
       depending on the size of original_fasta"""
    #
    # Clean up any segment files from previous runs before creating new one
    #
    for i in segments:
        os.unlink(i)
    #
    current_file_index = 0
    for line in open(original_fasta):
        #
        # start a new file for each accession line
        #
        if line[0] == '>':
            current_file_index += 1
            file_name = "%d.segment" % current_file_index
            file_path = os.path.join(temp_directory, file_name)
            current_file = open(file_path, "w")
            current_file.write(line)

@transform(splitFasta, suffix(".segment"), [".blastResult", ".blastSuccess"])
def runBlast(seqFile, output_files):
    #
    blastResultFile, flag_file = output_files
    #
    run_cmd("blastall -p blastp -d human.protein.faa -i %s > %s" % (seqFile, blastResultFile))
    #
    # "touch" flag file to indicate success
    #
    open(flag_file, "w")

@merge(runBlast, result_file)
def combineBlastResults (blastResult_and_flag_Files, combinedBlastResultFile):
    """Combine blast results"""
    #
    output_file = open(combinedBlastResultFile, "w")
    for blastResult_file, flag_file in blastResult_and_flag_Files:
        output_file.write(open(blastResult_file).read())

```

### 3.5 Code for the manual tutorial

- The *\*\*Ruffus\*\** manual

### 3.6 Code for the simple tutorial: 8 steps to *Ruffus*

- A simple tutorial

### 3.6.1 Table of Contents

*Chain tasks (functions) together into a pipeline Provide parameters to run jobs in parallel Tracing through your new pipeline Using flowcharts Split up a large problem into smaller chunks Calculate partial solutions in parallel Re-combine the partial solutions into the final result Automatically signal the completion of each step of our pipeline*

REFERENCE:

## 4.1 Decorators

### 4.1.1 Ruffus Decorators

See also:

*Indicator objects*

*Core*

Decorator	Examples	
<code>@originate</code> ( <i>Summary / Manual</i> ) <ul style="list-style-type: none"><li>Creates (originates) a set of starting file without dependencies from scratch (<i>ex nihilo!</i>)</li><li>Only called to create files which do not exist.</li><li>Invoked once (a job created) per item in the <code>output_files</code> list.</li></ul>	<ul style="list-style-type: none"><li><code>@originate ( output_files, [extra_parameters,...] )</code></li></ul>	
<code>@split</code> ( <i>Summary / Manual</i> ) <ul style="list-style-type: none"><li>Splits a single input into multiple output</li><li>Globs in <code>output</code> can specify an indeterminate number of files.</li></ul>	<ul style="list-style-type: none"><li><code>@split ( tasks_or_file_names, output_files, [extra_parameters,...] )</code></li></ul>	
<code>@transform</code> ( <i>Summary / Manual</i> ) <ul style="list-style-type: none"><li>Applies the task function to transform input data to output.</li></ul>	<ul style="list-style-type: none"><li><code>@transform ( tasks_or_file_names, suffix(suffix_string), output_path )</code></li><li><code>@transform ( tasks_or_file_names, regex(regex_pattern), output_path )</code></li><li><code>@transform ( tasks_or_file_names, formatter(regex_pattern), output_path )</code></li></ul>	
<code>@merge</code> ( <i>Summary / Manual</i> ) <ul style="list-style-type: none"><li>Merges multiple input files into a single output.</li></ul>	<ul style="list-style-type: none"><li><code>@merge ( tasks_or_file_names, output, [extra_parameters,...] )</code></li></ul>	



## Combinatorics

Decorator	Examples
<code>@product</code> ( <i>Summary / Manual</i> ) <ul style="list-style-type: none"> <li>Generates the <b>product</b>, i.e. all vs all comparisons, between sets of input files.</li> </ul>	<ul style="list-style-type: none"> <li><code>@product ( tasks_or_file_names,formatter([ regex_pattern ]),[* task)</code></li> </ul>
<code>@permutations</code> ( <i>Summary / Manual</i> ) <ul style="list-style-type: none"> <li>Generates the <b>permutations</b>, between all the elements of a set of <b>Input</b></li> <li>Analogous to the python <code>itertools.permutations</code></li> <li><code>permutations('ABCD', 2) -&gt; AB AC AD BA BC BD CA CB CD DA DB DC</code></li> </ul>	<ul style="list-style-type: none"> <li><code>@permutations ( tasks_or_file_names,formatter([ regex_pattern ]),tuple)</code></li> </ul>
<code>@combinations</code> ( <i>Summary / Manual</i> ) <ul style="list-style-type: none"> <li>Generates the <b>permutations</b>, between all the elements of a set of <b>Input</b></li> <li>Analogous to the python <code>itertools.combinations</code></li> <li><code>combinations('ABCD', 3) -&gt; ABC ABD ACD BCD</code></li> <li>Generates the <b>combinations</b>, between all the elements of a set of <b>Input</b>: i.e. r-length tuples of <i>input</i> elements with no repeated elements (<b>A A</b>) and where order of the tuples is irrelevant (either <b>A B</b> or <b>B A</b>, not both).</li> </ul>	<ul style="list-style-type: none"> <li><code>@combinations ( tasks_or_file_names,formatter([ regex_pattern ]),tuple)</code></li> </ul>
<code>@combinations_with_replacement</code> ( <i>Summary / Manual</i> ) <ul style="list-style-type: none"> <li>Generates the <b>permutations</b>, between all the elements of a set of <b>Input</b></li> <li>Analogous to the python <code>itertools.permutations</code></li> <li><code>combinations('ABCD', 3) -&gt; ABC ABD ACD BCD</code></li> <li>Generates the <b>combinations_with_replacement</b>, between all the elements of a set of <b>Input</b>: i.e. r-length tuples of <i>input</i> elements with no repeated elements (<b>A A</b>) and where order of the tuples is irrelevant (either <b>A B</b> or <b>B A</b>, not both).</li> </ul>	<ul style="list-style-type: none"> <li><code>@combinations_with_replacement ( tasks_or_file_names,formatter([ regex_pattern ]),tuple)</code></li> </ul>



## Advanced

Decorator	Examples
<p><b>@subdivide</b> (<i>Summary / Manual</i>) - Subdivides a set of <i>Inputs</i> each further into multiple <i>Outputs</i>. - The number of files in each <i>Output</i> can be set at runtime by the use of globs. - <b>Many to Even More</b> operator. - The use of <b>split</b> is a synonym for subdivide is deprecated.</p>	<ul style="list-style-type: none"> <li>• <code>@subdivide ( tasks_or_file_names, regex(regex_pattern), [ inputs   add ] )</code></li> <li>• <code>@subdivide ( tasks_or_file_names, formatter([regex_pattern]), [ inputs   add ] )</code></li> </ul>
<p><b>@transform</b> (<i>Summary / Manual</i>)</p> <ul style="list-style-type: none"> <li>• Infers input as well as output from regular expression substitutions</li> <li>• Useful for adding additional file dependencies</li> </ul>	<ul style="list-style-type: none"> <li>• <code>@transform ( tasks_or_file_names, regex(regex_pattern), [ inputs   add ] )</code></li> <li>• <code>@transform ( tasks_or_file_names, formatter(regex_pattern), [ inputs   add ] )</code></li> </ul>
<p><b>@collate</b> (<i>Summary / Manual</i>)</p> <ul style="list-style-type: none"> <li>• Groups multiple input files using regular expression matching</li> <li>• Input resulting in the same output after substitution will be collated together.</li> </ul>	<ul style="list-style-type: none"> <li>• <code>@collate ( tasks_or_file_names, regex(regex_pattern), output_pattern )</code></li> <li>• <code>@collate ( tasks_or_file_names, regex(regex_pattern), inputs   add_input )</code></li> <li>• <code>@collate ( tasks_or_file_names, formatter(formatter_pattern), output_pattern )</code></li> <li>• <code>@collate ( tasks_or_file_names, formatter(formatter_pattern), inputs   add_input )</code></li> </ul>
<p><b>@follows</b> (<i>Summary / Manual</i>)</p> <ul style="list-style-type: none"> <li>• Indicates task dependency</li> <li>• optional <i>mkdir</i> prerequisite (<i>see Manual</i>)</li> </ul>	<ul style="list-style-type: none"> <li>• <code>@follows ( task1, 'task2' )</code></li> <li>• <code>@follows ( task1, mkdir( 'my/directory/' ) )</code></li> </ul>
<p><b>@posttask</b> (<i>Summary / Manual</i>)</p> <ul style="list-style-type: none"> <li>• Calls function after task completes</li> <li>• Optional <i>touch_file</i> indicator (<i>Manual</i>)</li> </ul>	<ul style="list-style-type: none"> <li>• <code>@posttask ( signal_task_completion_function )</code></li> <li>• <code>@posttask ( touch_file( 'task1.completed' ) )</code></li> </ul>
<p><b>@active_if</b> (<i>Summary / Manual</i>)</p> <ul style="list-style-type: none"> <li>• Switches tasks on and off at run time depending on its parameters</li> <li>• Evaluated each time <i>pipeline_run(...)</i>, <i>pipeline_printout(...)</i> or <i>pipeline_printout_graph(...)</i> is called.</li> <li>• Dormant tasks behave as if they are up to date and have no output.</li> </ul>	<ul style="list-style-type: none"> <li>• <code>@active_if ( on_or_off1, [on_or_off2, ...] )</code></li> </ul>
<p><b>@jobs_limit</b> (<i>Summary / Manual</i>)</p>	<ul style="list-style-type: none"> <li>• <code>@jobs_limit ( NUMBER_OF_JOBS_RUNNING_CONCURRENTLY )</code></li> </ul>
<b>4.1. Decorators</b> <ul style="list-style-type: none"> <li>• Limits the amount of multi-processing for the specified task</li> <li>• Ensures that fewer than N</li> </ul>	213

**Esoteric!**

Decorator	Examples
<code>@files</code> ( <i>Summary / Manual</i> ) <ul style="list-style-type: none"><li>• I/O parameters</li><li>• skips up-to-date jobs</li><li>• Should use <code>@transform</code> etc instead</li></ul>	<ul style="list-style-type: none"><li>• <code>@files( parameter_list )</code></li><li>• <code>@files( parameter_generating_function )</code></li><li>• <code>@files( input_file, output_file, other_params, ... )</code></li></ul>
<code>@parallel</code> ( <i>Summary / Manual</i> ) <ul style="list-style-type: none"><li>• By default, does not check if jobs are up to date</li><li>• Best used in conjunction with <code>@check_if_upToDate</code></li></ul>	<ul style="list-style-type: none"><li>• <code>@parallel( parameter_list )</code> (<i>see Manual</i>)</li><li>• <code>@parallel( parameter_generating_function )</code> (<i>see Manual</i>)</li></ul>
<code>@check_if_upToDate</code> ( <i>Summary / Manual</i> ) <ul style="list-style-type: none"><li>• Custom function to determine if jobs need to be run</li></ul>	<ul style="list-style-type: none"><li>• <code>@check_if_upToDate( is_task_up_to_date_function )</code></li></ul>
<b>Tip:</b> The use of this overly complicated function is discouraged. <code>@files_re</code> ( <i>Summary</i> ) <ul style="list-style-type: none"><li>• I/O file names via regular expressions</li><li>• start from lists of file names or <code>glob</code> results</li><li>• skips up-to-date jobs</li></ul>	<ul style="list-style-type: none"><li>• <code>@files_re( tasks_or_file_names, matching_regex, [input_pattern,] output_pattern )</code> are regex patterns used to create input/output file names from the starting list of either <code>glob_str</code> or file names</li></ul>

See also:

*Decorators*

#### 4.1.2 Indicator Objects

How *ruffus* disambiguates certain parameters to decorators.

They are like `keyword arguments` in python, a little more verbose but they make the syntax much simpler.

Indicator objects are also “self-documenting” so you can see exactly what is happening clearly.

#### *formatter*

`formatter([ regex | None , regex | None... ])`

- The optional enclosed parameters are a python regular expression strings
- Each regular expression matches a corresponding *Input* file name string
- *formatter* parses each file name string into path and regular expression components
- Parsing fails altogether if the regular expression is not matched

Path components include:

- basename: The *base name excluding extension*, "file.name"
- ext : The *extension*, ".ext"
- path : The *dirname*, "/directory/to/a"
- subdir : A list of sub-directories in the path in reverse order, ["a", "to", "directory", "/"]
- subpath : A list of descending sub-paths in reverse order, ["/directory/to/a", "/directory/to", "/directory", "/"]

The replacement string refers to these components using python `string.format` style curly braces. {NAME}

We refer to an element from the Nth input string by index, for example:

- "{ext [0]}" is the extension of the first input string.
- "{basename [1]}" is the basename of the second input string.
- "{basename [1] [0:3]}" are the first three letters from the basename of the second input string.

#### Used by:

- @split
- @transform
- @merge
- @subdivide
- @collate
- @product
- @permutations
- @combinations
- @combinations\_with\_replacement

#### @transform example:

```
from ruffus import *

#    create initial file pairs
@originate([
    ['job1.a.start', 'job1.b.start'],
    ['job2.a.start', 'job2.b.start'],
    ['job3.a.start', 'job3.c.start']    ])
def create_initial_file_pairs(output_files):
    for output_file in output_files:
        with open(output_file, "w") as oo: pass

#-----
#    formatter
#
@transform(create_initial_file_pairs,          # Input
          formatter(".+/job(?P<JOBNUMBER>\d+).a.start",
                    ".+/job[123].b.start"),      # Extract job number
          # Match only "b" files
```

```
[ "{path[0]}/jobs{JOBNUMBER[0]}.output.a.1",
    "{path[1]}/jobs{JOBNUMBER[0]}.output.b.1"]" # Replacement list
def first_task(input_files, output_parameters):
    print "input_parameters = ", input_files
    print "output_parameters = ", output_parameters

#
#       Run
#
pipeline_run(verbose=0)
```

This produces:

```
input_parameters = ['job1.a.start',
                    'job1.b.start']
output_parameters = ['/home/lg/src/temp/jobs1.output.a.1',
                     '/home/lg/src/temp/jobs1.output.b.1', 45]

input_parameters = ['job2.a.start',
                    'job2.b.start']
output_parameters = ['/home/lg/src/temp/jobs2.output.a.1',
                     '/home/lg/src/temp/jobs2.output.b.1', 45]
```

#### @permutations example:

Combinatoric decorators such as `@product` or `@product` behave much like nested for loops in enumerating, combining, and permutating the original sets of inputs.

The replacement strings require an extra level of indirection to refer to parsed components:

```
from ruffus import *
from ruffus.combinatorics import *

#   create initial files
@originate(['a.start', 'b.start', 'c.start'])
def create_initial_files(output_file):
    with open(output_file, "w") as oo: pass

#-----
#   formatter
#
@permutations(create_initial_files, # Input

              formatter("(start)$"),
              2, # match input

              "{path[0][0]}/{basename[0][0]}_vs_{basename[1][0]}.product",
              "{path[0][0]}",
              ["{basename[0][0]}",
               "{basename[1][0]}"])

def product_task(input_file, output_parameter, shared_path, basenames):
    print "input_parameter = ", input_file
    print "output_parameter = ", output_parameter
    print "shared_path      = ", shared_path
    print "basenames        = ", basenames
```

---

```

#
#      Run
#
pipeline_run(verbose=0)

```

This produces:

```

>>> pipeline_run(verbose=0)
input_parameter = ('a.start', 'b.start')
output_parameter = /home/lg/src/oss/ruffus/a_vs_b.product
shared_path      = /home/lg/src/oss/ruffus
basenames        = ['a', 'b']

input_parameter = ('a.start', 'c.start')
output_parameter = /home/lg/src/oss/ruffus/a_vs_c.product
shared_path      = /home/lg/src/oss/ruffus
basenames        = ['a', 'c']

input_parameter = ('b.start', 'a.start')
output_parameter = /home/lg/src/oss/ruffus/b_vs_a.product
shared_path      = /home/lg/src/oss/ruffus
basenames        = ['b', 'a']

input_parameter = ('b.start', 'c.start')
output_parameter = /home/lg/src/oss/ruffus/b_vs_c.product
shared_path      = /home/lg/src/oss/ruffus
basenames        = ['b', 'c']

input_parameter = ('c.start', 'a.start')
output_parameter = /home/lg/src/oss/ruffus/c_vs_a.product
shared_path      = /home/lg/src/oss/ruffus
basenames        = ['c', 'a']

input_parameter = ('c.start', 'b.start')
output_parameter = /home/lg/src/oss/ruffus/c_vs_b.product
shared_path      = /home/lg/src/oss/ruffus
basenames        = ['c', 'b']

```

## **suffix**

**suffix(string)**

The enclosed parameter is a string which must match *exactly* to the end of a file name.

**Used by:**

- @transform

**Example:**

```

#
#      Transforms '*.c' to '*.o':::
#
@transform(previous_task, suffix(".c"), ".o")
def compile(infile, outfile):
    pass

```

## regex

```
regex( regular_expression )
```

The enclosed parameter is a python regular expression string, which must be wrapped in a `regex` indicator object.

See python [regular expression \(re\)](#) documentation for details of regular expression syntax

### Used by:

- `@transform`
- `@subdivide`
- `@collate`
- The deprecated `@files_re`

### Example:

```
@transform(previous_task, regex(r".c$"), ".o")
def compile(infile, outfile):
    pass
```

## add\_inputs

```
add_inputs( input_file_pattern )
```

The enclosed parameter(s) are pattern strings or a nested structure which is added to the input for each job.

### Used by:

- `@transform`
- `@collate`
- `@subdivide`

### Example @transform with suffix(...)

A common task in compiling C code is to include the corresponding header file for the source.  
To compile \*.c to \*.o, adding \*.h and the common header universal.h:

```
@transform(["1.c", "2.c"], suffix(".c"), add_inputs([r"\1.h", "universal.h"]), ".o")
def compile(infile, outfile):
    # do something here
    pass
```

The starting file names are 1.c and 2.c.

`suffix(".c")` matches ".c" so \1 stands for the unmatched prefixes "1" and "2"

### This will result in the following functional calls:

```
compile(["1.c", "1.h", "universal.h"], "1.o")
compile(["2.c", "2.h", "universal.h"], "2.o")
```

A string like `universal.h` in `add_inputs` will be added *as is*. `r"\1.h"`, however, performs suffix substitution, with the special form `r"\1"` matching everything up to the suffix. Remember to ‘escape’ `r"\1"` otherwise Ruffus will complain and throw an `Exception` to remind you. The most convenient way is to use a python “raw” string.

#### Example of `add_inputs(...)` with `regex(...)`

The suffix match (`suffix(...)`) is exactly equivalent to the following code using regular expression (`regex(...)`)

```
@transform(["1.c", "2.c"], regex(r"^(.+)\.c$"), add_inputs([r"\1.h", "universal.h"]))
def compile(infile, outfile):
    # do something here
    pass
```

The `suffix(...)` code is much simpler but the regular expression allows more complex substitutions.

#### `add_inputs(...)` preserves original inputs

`add_inputs` nests the the original input parameters in a list before adding additional dependencies.

This can be seen in the following example:

```
@transform([
    ["1.c", "A.c", 2],
    ["2.c", "B.c", "C.c", 3]],
    suffix(".c"), add_inputs([r"\1.h", "universal.h"]), ".o")
def compile(infile, outfile):
    # do something here
    pass
```

This will result in the following functional calls:

```
compile([[1.c, "A.c", 2], "1.h", "universal.h"], "1.o")
compile([[3.c, "B.c", "C.c", 3], "2.h", "universal.h"], "2.o")
```

The original parameters are retained unchanged as the first item in a list

## *inputs*

`inputs(input_file_pattern)`

Used by:

- `@transform`
- `@collate`
- `@subdivide`

The enclosed single parameter is a pattern string or a nested structure which is used to construct the input for each job.

If more than one argument is supplied to `inputs`, an exception will be raised.

Use a tuple or list (as in the following example) to send multiple input arguments to each job.

Used by:

- The advanced form of `@transform`

### inputs(...) replaces original inputs

`inputs(...)` allows the original input parameters to be replaced wholesale.

This can be seen in the following example:

```
@transform([
    ["1.c", "A.c", 2],
    ["2.c", "B.c", "C.c", 3]],
    suffix(".c"),
    inputs([r"1.py", "docs.rst"]),
    ".pyc")
def compile(infile, outfile):
    # do something here
    pass
```

This will result in the following functional calls:

```
compile(["1.py", "docs.rst"], "1.pyc")
compile(["2.py", "docs.rst"], "2.pyc")
```

In this example, the corresponding python files have been sneakily substituted without trace in the place of the C source files.

### `mkdir`

`mkdir(directory_name1, [directory_name2, ...])`

The enclosed parameter is a directory name or a sequence of directory names. These directories will be created as part of the prerequisites of running a task.

Used by:

- `@follows`

Example:

```
@follows(mkdir("/output/directory"))
def task():
    pass
```

### `touch_file`

`touch_file(file_name)`

The enclosed parameter is a file name. This file will be `touch`-ed after a task is executed.

This will change the date/time stamp of the `file_name` to the current date/time. If the file does not exist, an empty file will be created.

Used by:

- `@posttask`

Example:

```
@posttask(touch_file("task_completed.flag"))
@files(None, "a.1")
def do_task(input_file, output_file):
    pass
```

***output\_from***

```
output_from( file_name_string1 [, file_name_string1 ,...])
```

Indicates that any enclosed strings are not file names but refer to task functions.

**Used by:**

- *@split*
- *@transform*
- *@merge*
- *@collate*
- *@subdivide*
- *@product*
- *@permutations*
- *@combinations*
- *@combinations\_with\_replacement*
- *@files*

**Example:**

```
@split(["a.file", ("b.file", output_from("task1", 76, "task2"))], "*.split")
def task2(input, output):
    pass
```

is equivalent to:

```
@split(["a.file", ("b.file", (task1, 76, task2))], "*.split")
def task2(input, output):
    pass
```

***combine***

```
combine(arguments)
```

**Warning:** This is deprecated syntax.

Please do not use!

*@merge* and *@collate* are more powerful and have straightforward syntax.

Indicates that the *inputs* of *@files\_re* will be collated or summarised into *outputs* by category. See the *Manual* or :ref:`@collate <new\_manual.collate>` for examples.

**Used by:**

- *@files\_re*

**Example:**

```
@files_re('*.animals',                                     # inputs = all *.animal files
          r'mammals.([^.]+)',                         # regular expression
          combine(r'\1/animals.in_my_zoo'),             # single output file per species
          r'\1' )                                         # species name
def capture_mammals(infiles, outfile, species):
```

```
# summarise all animals of this species
"""


```

**Core****See also:**

- *Decorators* for more decorators

**4.1.3 @originate****@originate ( *output\_files*, [*extra\_parameters*,...] )****Purpose:**

- Creates (originates) a set of starting file without dependencies from scratch (*ex nihilo!*)
- Only called to create files which do not exist.
- Invoked once (a job created) per item in the *output\_files* list.

---

**Note:** The first argument for the task function is the *Output*. There is by definition no *Input* for @originate

---

**Example:**

```
from ruffus import *
@originate(["a", "b", "c", "d"], "extra")
def test(output_file, extra):
    open(output_file, "w")

pipeline_run()

>>> pipeline_run()
Job = [None -> a, extra] completed
Job = [None -> b, extra] completed
Job = [None -> c, extra] completed
Job = [None -> d, extra] completed
Completed Task = test

>>> # all files exist: nothing to do
>>> pipeline_run()

>>> # delete 'a' so that it is missing
>>> import os
>>> os.unlink("a")

>>> pipeline_run()
Job = [None -> a, extra] completed
Completed Task = test
```

**Parameters:**

- *output\_files*
  - Can be a single file name or a list of files
  - Each item in the list is treated as the *Output* of a separate job
- *extra\_parameters* Any extra parameters are passed verbatim to the task function

**See also:**

- *Decorators* for more decorators

**4.1.4 @split****@split ( *tasks\_or\_file\_names*, *output\_files*, [*extra\_parameters*,...] )****Purpose:**

Splits a single set of input files into multiple output file names, where the number of output files may not be known beforehand.

Only out of date tasks (comparing input and output files) will be run

**4.1. Decorators**

223

```
@split("big_file", '*.little_files')
def split_big_to_small(input_file, output_files):
    print "input_file = %s" % input_file
```



**For advanced users****See also:**

- *Decorators* for more decorators

**4.1.8 @subdivide**

**@subdivide ( tasks\_or\_file\_names, regex(matching\_regex) | formatter(matching\_formatter), [ inputs (input\_pattern\_or\_glob) | add\_inputs (input\_pattern\_or\_glob) ], output\_pattern, [extra\_parameters,...] )**

**Purpose:**

- Subdivides a set of *Inputs* each further into multiple *Outputs*.
- **Many to Even More** operator
- The number of files in each *Output* can be set at runtime by the use of globs
- Output file names are specified using the *formatter* or *regex* indicators from *tasks\_or\_file\_names*, i.e. from the output of specified tasks, or a list of file names, or a *glob* matching pattern.
- **Additional inputs or dependencies can be added dynamically to the task:**
  - add\_inputs* nests the the original input parameters in a list before adding additional dependencies.
  - inputs* replaces the original input parameters wholesale.
- Only out of date tasks (comparing input and output files) will be run.

---

**Note:** The use of **split** is a synonym for **subdivide** is deprecated.

---

**Example:**

```
from ruffus import *
from random import randint
from random import os

@originate(['0.start', '1.start', '2.start'])
def create_files(output_file):
    with open(output_file, "w"):
        pass

    #
    # Subdivide each of 3 start files further into [NNN1, NNN2, NNN3] number of files
    # where NNN1, NNN2, NNN3 are determined at run time
    #
    @subdivide(create_files, formatter(),
               "{path[0]}/{basename[0]}.*.step1", # Output parameter: Glob matches any number
               "{path[0]}/{basename[0]}")          # Extra parameter: Append to this for output
def subdivide_files(input_file, output_files, output_file_name_root):
    #
    # IMPORTANT: cleanup rubbish from previous run first
    #
    for oo in output_files:
        os.unlink(oo)
    # The number of output files is decided at run time
    number_of_output_files = randint(2,4)
    for ii in range(number_of_output_files):
        output_file_name = "{output_file_name_root}.{ii}.step1".format(**locals())
        with open(output_file_name, "w"):
            pass

    #

4.1. Decorators
```

Each output of `subdivide_files` results in a separate job for downstream tasks

---

```
# @transform(subdivide_files, suffix(".step1"), ".step2")
def analyse_files(input_file, output_file_name):
    with open(output_file_name, "w"):
```



## Combinatorics

### See also:

- *Decorators* for more decorators

### 4.1.17 @product

**@product ( tasks\_or\_file\_names, formatter(matching\_formatter), [tasks\_or\_file\_names, formatter(matching\_formatter), ... ], output\_pattern, [extra\_parameters,...] )**

#### Purpose:

Generates the Cartesian **product**, i.e. all vs all comparisons, between sets of input files.

The effect is analogous to the python `itertools` function of the same name, i.e. a nested for loop.

```
>>> from itertools import product
>>> # product('ABC', 'XYZ') --> AX AY AZ BX BY BZ CX CY CZ
>>> [ "" .join(a) for a in product('ABC', 'XYZ') ]
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

Only out of date tasks (comparing input and output files) will be run

Output file names and strings in the extra parameters are determined from `tasks_or_file_names`, i.e. from the output of up stream tasks, or a list of file names, after string replacement via `formatter`.

The replacement strings require an extra level of indirection to refer to parsed components:

1. The first level refers to which *set* of inputs (e.g. A,B or P,Q or X,Y in the following example.)

2. The second level refers to which input file in any particular *set* of inputs.

For example, '`{basename[2][0]}`' is the `basename` for

- the third set of inputs (X,Y) and
- the first file name string in each **Input** of that set ("x.1\_start" and "y.1\_start")

#### Example:

Calculates the @product of A,B and P,Q and X, Y files

```
from ruffus import *
from ruffus.combinatorics import *

# Three sets of initial files
@originate([ 'a.start', 'b.start' ])
def create_initial_files_ab(output_file):
    with open(output_file, "w") as oo: pass

@originate([ 'p.start', 'q.start' ])
def create_initial_files_pq(output_file):
    with open(output_file, "w") as oo: pass

@originate([ ['x.1_start', 'x.2_start'],
             ['y.1_start', 'y.2_start'] ])
def create_initial_files_xy(output_file):
    with open(output_file, "w") as oo: pass

# @product
@product(   create_initial_files_ab,           # Input
            formatter("(..start)$"),          # match input file set # 1

            create_initial_files_pq,         # Input
            formatter("(..start)$"),          # match input file set # 2

            create_initial_files_xy,         # Input
            formatter("{path[0][0]} / {basename[0][0]}_vs_",
                      "# Output Replacement string") # match input file set # 3)
```



**Esoteric****See also:**

- *Decorators* for more decorators

**4.1.21 Generating parameters on the fly for @files****@files (custom\_function)****Purpose:**

Uses a custom function to generate sets of parameters to separate jobs which can run in parallel.

The first two parameters in each set represent the input and output which are used to see if the job is out of date and needs to be (re-)run.

By default, out of date checking uses input/output file timestamps. (On some file systems, timestamps have a resolution in seconds.) See `@check_if_upToDate()` for alternatives.

**Example:**

```
from ruffus import *
def generate_parameters_on_the_fly():
    parameters = [
        ['input_file1', 'output_file1', 1, 2], # 1st job
        ['input_file2', 'output_file2', 3, 4], # 2nd job
        ['input_file3', 'output_file3', 5, 6], # 3rd job
    ]
    for job_parameters in parameters:
        yield job_parameters

@files(generate_parameters_on_the_fly)
def parallel_io_task(input_file, output_file, param1, param2):
    pass

pipeline_run([parallel_task])
```

is the equivalent of calling:

```
parallel_io_task('input_file1', 'output_file1', 1, 2)
parallel_io_task('input_file2', 'output_file2', 3, 4)
parallel_io_task('input_file3', 'output_file3', 5, 6)
```

**Parameters:**

- *custom\_function*: Generator function which yields each time a complete set of parameters for one job

**Checking if jobs are up to date:** Strings in `input` and `output` (including in nested sequences) are interpreted as file names and used to check if jobs are up-to-date.

See *above* for more details

**See also:**

- *Decorators* for more decorators

**4.1.22 @check\_if\_upToDate****@check\_if\_upToDate (dependency\_checking\_function)**

**Purpose:** Checks to see if a job is up to date, and needs to be run.

Usually used in conjunction with `@parallel()`

**Example:**

```
from ruffus import *
import os
def check_file_exists(input_file, output_file):
    if not os.path.exists(output_file):
        return True, "Missing file %s" % output_file
    else:
        return False, "File %s exists" % output_file

@parallel([[None, "a.1"]])
```



**Deprecated****See also:**

- *Decorators* for more decorators

**4.1.24 @files****@files (input1, output1, [extra\_parameters1, ...])****@files for single jobs****Purpose:** Provides parameters to run a task.

The first two parameters in each set represent the input and output which are used to see if the job is out of date and needs to be (re-)run.

By default, out of date checking uses input/output file timestamps. (On some file systems, timestamps have a resolution in seconds.) See `@check_if_upToDate()` for alternatives.

**Example:**

```
from ruffus import *
@files('a.1', 'a.2', 'A file')
def transform_files(infile, outfile, text):
    pass
pipeline_run([transform_files])
```

If `a.2` is missing or was created before `a.1`, then the following will be called:

```
transform_files('a.1', 'a.2', 'A file')
```

**Parameters:**

- *input* Input file names
- *output* Output file names
  - *extra\_parameters* optional extra\_parameters are passed verbatim to each job.

**Checking if jobs are up to date:** Strings in `input` and `output` (including in nested sequences) are interpreted as file names and used to check if jobs are up-to-date.

See *above* for more details

**@files ( (( input, output, [extra\_parameters,...] ), (...), ... ) )****@files in parallel****Purpose:**

Passes each set of parameters to separate jobs which can run in parallel

The first two parameters in each set represent the input and output which are used to see if the job is out of date and needs to be (re-)run.

By default, out of date checking uses input/output file timestamps. (On some file systems, timestamps have a resolution in seconds.) See `@check_if_upToDate()` for alternatives.

**Example:**

```
from ruffus import *
parameters = [
    [ 'a.1', 'a.2', 'A file' ], # 1st job
    [ 'b.1', 'b.2', 'B file' ], # 2nd job
]

@files(parameters)
def parallel_io_task(infile, outfile, text):
    pass
pipeline_run([parallel_io_task])
```

is the equivalent of calling:

```
parallel_io_task('a.1', 'a.2', 'A file')
```

**Parameters:**

- *input* Input file names
- *output* Output file names

## 4.2 Modules:

### 4.2.1 ruffus.Task

#### ruffus.task – Overview

Initial implementation of @active\_if by Jacob Biesinger

##### Decorator syntax:

Pipelined tasks are created by “decorating” a function with the following syntax:

```
def func_a():
    pass

@follows(func_a)
def func_b():
    pass
```

Each task is a single function which is applied one or more times to a list of parameters (typically input files to produce a list of output files).

Each of these is a separate, independent job (sharing the same code) which can be run in parallel.

#### Running the pipeline

To run the pipeline:

```
pipeline_run(target_tasks, forcedtorun_tasks = [],
            multiprocess = 1,
            logger = stderr_logger,
            gnu_make_maximal_rebuild_mode = True,
            cleanup_log = "../cleanup.log")

pipeline_cleanup(cleanup_log = "../cleanup.log")
```

#### Decorators

Basic Task decorators are:

*@follows()*  
and  
*@files()*

Task decorators include:

*@split()*  
*@transform()*  
*@merge()*  
*@posttask()*

More advanced users may require:

```
@transform()  
@collate()  
@parallel()  
@check_if_upToDate()  
@files_re()
```

## Pipeline functions

### `pipeline_run`

```
ruffus.task.pipeline_run(target_tasks,      forcedtorun_tasks=[ ],      multiprocess=1,      log-  
ger=stderr_logger, gnu_make_maximal_rebuild_mode=True)
```

Run pipelines.

#### Parameters

- **target\_tasks** – targets task functions which will be run if they are out-of-date
- **forcedtorun\_tasks** – task functions which will be run whether or not they are out-of-date
- **multiprocess** – The number of concurrent jobs running on different processes.
- **multithread** – The number of concurrent jobs running as different threads. If > 1, ruffus will use multithreading *instead of* multiprocessing (and ignore the multiprocess parameter). Using multi threading is particularly useful to manage high performance clusters which otherwise are prone to “processor storms” when large number of cores finish jobs at the same time. (Thanks Andreas Heger)
- **logger** ([logging objects](#)) – Where progress will be logged. Defaults to stderr output.
- **verbose** – level 0 : nothing level 1 : Out-of-date Tasks (names and warnings) level 2 : All Tasks (including any task function docstrings) level 3 : Out-of-date Jobs in Out-of-date Tasks, no explanation level 4 : Out-of-date Jobs in Out-of-date Tasks, with explanation level 5 : All Jobs in Out-of-date Tasks, (include only list of up-to-date tasks) level 6 : All jobs in All Tasks whether out of date or not level 10: logs messages useful only for debugging ruffus pipeline code
- **touch\_files\_only** – Create or update input/output files only to simulate running the pipeline. Do not run jobs. If set to CHECKSUM\_REGENERATE, will regenerate the checksum history file to reflect the existing i/o files on disk.
- **exceptions\_terminate\_immediately** – Exceptions cause immediate termination rather than waiting for N jobs to finish where N = multiprocess
- **log\_exceptions** – Print exceptions to the logger as soon as they occur.
- **checksum\_level** – Several options for checking up-to-dateness are available: Default is level 1. level 0 : Use only file timestamps level 1 : above, plus timestamp of successful job completion level 2 : above, plus a checksum of the pipeline function body level 3 : above, plus a checksum of the pipeline function default arguments and the additional arguments passed in by task decorators
- **history\_file** – The database file which stores checksums and file timestamps for input/output files.

- **one\_second\_per\_job** – To work around poor file timestamp resolution for some file systems. Defaults to True if checksum\_level is 0 forcing Tasks to take a minimum of 1 second to complete.
- **runtime\_data** – Experimental feature for passing data to tasks at run time
- **gnu\_make\_maximal\_rebuild\_mode** – Defaults to re-running *all* out-of-date tasks. Runs minimal set to build targets if set to True. Use with caution.

## **pipeline\_printout**

```
ruffus.task.pipeline_printout(output_stream=None, target_tasks=[], forcedtorun_tasks=[],
                           verbose=1, indent=4, gnu_make_maximal_rebuild_mode=True,
                           wrap_width=100, runtime_data=None, checksum_level=None,
                           history_file=None)
```

Printouts the parts of the pipeline which will be run

Because the parameters of some jobs depend on the results of previous tasks, this function produces only the current snap-shot of task jobs. In particular, tasks which generate variable number of inputs into following tasks will not produce the full range of jobs.

```
verbose = 0 : nothing
verbose = 1 : print task name
verbose = 2 : print task description if exists
verbose = 3 : print job names for jobs to be run
verbose = 4 : print list of up-to-date tasks and job names for jobs to be run
verbose = 5 : print job names for all jobs whether up-to-date or not
```

### **Parameters**

- **output\_stream** (file-like object with `write()` function) – where to print to
- **target\_tasks** – targets task functions which will be run if they are out-of-date
- **forcedtorun\_tasks** – task functions which will be run whether or not they are out-of-date
- **verbose** – level 0 : nothing level 1 : Out-of-date Tasks (names and warnings) level 2 : All Tasks (including any task function docstrings) level 3 : Out-of-date Jobs in Out-of-date Tasks, no explanation level 4 : Out-of-date Jobs in Out-of-date Tasks, with explanation level 5 : All Jobs in Out-of-date Tasks, (include only list of up-to-date tasks) level 6 : All jobs in All Tasks whether out of date or not level 10: logs messages useful only for debugging ruffus pipeline code
- **indent** – How much indentation for pretty format.
- **gnu\_make\_maximal\_rebuild\_mode** – Defaults to re-running *all* out-of-date tasks. Runs minimal set to build targets if set to True. Use with caution.
- **wrap\_width** – The maximum length of each line
- **runtime\_data** – Experimental feature for passing data to tasks at run time
- **checksum\_level** – Several options for checking up-to-dateness are available: Default is level 1. level 0 : Use only file timestamps level 1 : above, plus timestamp of successful job completion level 2 : above, plus a checksum of the pipeline function body level 3 : above, plus a checksum of the pipeline function default arguments and the additional arguments passed in by task decorators

## **pipeline\_printout\_graph**

```
ruffus.task.pipeline_printout_graph(stream,      output_format=None,      target_tasks=[  
    ],      forcedtorun_tasks=[],      draw_vertically=True,  
    ignore_upstream_of_target=False,  
    skip_uptodate_tasks=False,  
    gnu_make_maximal_rebuild_mode=True,  
    test_all_task_for_update=True,  no_key_legend=False,  
    minimal_key_legend=True, user_colour_scheme=None,  
    pipeline_name='Pipeline:', size=(11, 8), dpi=120,  
    runtime_data=None,    checksum_level=None,    his-  
    tory_file=None)
```

print out pipeline dependencies in various formats

### Parameters

- **stream** (file-like object with `write()` function) – where to print to
- **output\_format** – [“dot”, “jpg”, “svg”, “ps”, “png”]. All but the first depends on the `dot` program.
- **target\_tasks** – targets task functions which will be run if they are out-of-date.
- **forcedtorun\_tasks** – task functions which will be run whether or not they are out-of-date.
- **draw\_vertically** – Top to bottom instead of left to right.
- **ignore\_upstream\_of\_target** – Don’t draw upstream tasks of targets.
- **skip\_uptodate\_tasks** – Don’t draw up-to-date tasks if possible.
- **gnu\_make\_maximal\_rebuild\_mode** – Defaults to re-running *all* out-of-date tasks. Runs minimal set to build targets if set to `True`. Use with caution.
- **test\_all\_task\_for\_update** – Ask all task functions if they are up-to-date.
- **no\_key\_legend** – Don’t draw key/legend for graph.
- **checksum\_level** – Several options for checking up-to-dateness are available: Default is level 1. level 0 : Use only file timestamps level 1 : above, plus timestamp of successful job completion level 2 : above, plus a checksum of the pipeline function body level 3 : above, plus a checksum of the pipeline function default arguments and the additional arguments passed in by task decorators

## Logging

```
class ruffus.task.t_black_hole_logger  
    Does nothing!  
  
class ruffus.task.t_stderr_logger  
    Everything to stderr
```

## Implementation:

### Parameter factories:

```
ruffus.task.merge_param_factory(input_files_task_globs, output_param, *extra_params)  
Factory for task_merge
```

```
ruffus.task.collate_param_factory(input_files_task_globs, flatten_input, file_names_transform,  
                                  extra_input_files_task_globs,     replace_inputs,     out-  
                                  put_pattern, *extra_specs)
```

Factory for task\_collate

Looks exactly like @transform except that all [input] which lead to the same [output / extra] are combined together

```
ruffus.task.transform_param_factory(input_files_task_globs,                                    flatten_input,  
                                  file_names_transform,     extra_input_files_task_globs,  
                                  replace_inputs, output_pattern, *extra_specs)
```

Factory for task\_transform

```
ruffus.task.files_param_factory(input_files_task_globs,                                    flatten_input,  
                                  do_not_expand_single_job_tasks, output_extras)
```

**Factory for functions which** yield tuples of inputs, outputs / extras

..Note:

1. Each job requires input/output file names
2. Input/output file names can be a string, an arbitrarily nested sequence
3. Non-string types are ignored
3. Either Input or output file name must contain at least one string

```
ruffus.task.args_param_factory(orig_args)
```

**Factory for functions which** yield tuples of inputs, outputs / extras

..Note:

1. Each job requires input/output file names
2. Input/output file names can be a string, an arbitrarily nested sequence
3. Non-string types are ignored
3. Either Input or output file name must contain at least one string

```
ruffus.task.split_param_factory(input_files_task_globs,     output_files_task_globs,     *iex-  
                                  tra_params)
```

Factory for task\_split

### Wrappers around jobs:

```
ruffus.task.job_wrapper_generic(param,     user_defined_work_func,     register_cleanup,  
                                  touch_files_only)
```

run func

```
ruffus.task.job_wrapper_io_files(param,     user_defined_work_func,     register_cleanup,  
                                  touch_files_only, output_files_only=False)
```

run func on any i/o if not up to date

```
ruffus.task.job_wrapper_mkdir(param,     user_defined_work_func,     register_cleanup,  
                                  touch_files_only)
```

make directories if not exists

### Checking if job is update:

```
ruffus.task.needs_update_check_modify_time(*params, **kwargs)
```

Given input and output files, see if all exist and whether output files are later than input files Each can be

- 1.string: assumed to be a filename “file1”

- 2.any other type
- 3.arbitrary nested sequence of (1) and (2)

```
ruffus.task.needs_update_directory_missing(*params, **kwargs)
```

**Called per directory:** Does it exist? Is it an ordinary file not a directory? (throw exception)

## Exceptions and Errors

### 4.2.2 ruffus.proxy\_logger

#### Create proxy for logging for use with multiprocessing

These can be safely sent (marshalled) across process boundaries

##### Example 1

Set up logger from config file:

```
from proxy_logger import *
args={}
args["config_file"] = "/my/config/file"

(logger_proxy,
logging_mutex) = make_shared_logger_and_proxy (setup_std_shared_logger,
"my_logger", args)
```

##### Example 2

Log to file "/my/lg.log" in the specified format (Time / Log name / Event type / Message).

Delay file creation until first log.

Only log Debug messages

Other alternatives for the logging threshold (args ["level"]) include

- logging.DEBUG
- logging.INFO
- logging.WARNING
- logging.ERROR
- logging.CRITICAL

```
from proxy_logger import *
args={}
args["file_name"] = "/my/lg.log"
args["formatter"] = "%(asctime)s - %(name)s - %(levelname)6s - %(message)s"
args["delay"] = True
args["level"] = logging.DEBUG

(logger_proxy,
logging_mutex) = make_shared_logger_and_proxy (setup_std_shared_logger,
"my_logger", args)
```

### Example 3

Rotate log files every 20 Kb, with up to 10 backups.

```
from proxy_logger import *
args={}
args["file_name"] = "/my/lg.log"
args["rotating"] = True
args["maxBytes"] = 20000
args["backupCount"] = 10
(logger_proxy,
 logging_mutex) = make_shared_logger_and_proxy (setup_std_shared_logger,
 "my_logger", args)
```

#### To use:

```
(logger_proxy,
 logging_mutex) = make_shared_logger_and_proxy (setup_std_shared_logger,
 "my_logger", args)

with logging_mutex:
    my_log.debug('This is a debug message')
    my_log.info('This is an info message')
    my_log.warning('This is a warning message')
    my_log.error('This is an error message')
    my_log.critical('This is a critical error message')
    my_log.log(logging.DEBUG, 'This is a debug message')
```

Note that the logging function exception() is not included because python stack trace information is not well-marshalled ([pickled](#)) across processes.

### Proxies for a log:

ruffus.proxy\_logger.**make\_shared\_logger\_and\_proxy** (logger\_factory, logger\_name, args)  
Make a [logging](#) object called “logger\_name” by calling logger\_factory(args)

This function will return a proxy to the shared logger which can be copied to jobs in other processes, as well as a mutex which can be used to prevent simultaneous logging from happening.

#### Parameters

- **logger\_factory** – functions which creates and returns an object with the [logging](#) interface. setup\_std\_shared\_logger() is one example of a logger factory.
- **logger\_name** – name of log
- **args** – parameters passed (as a single argument) to logger\_factory

**Returns** a proxy to the shared logger which can be copied to jobs in other processes

**Returns** a mutex which can be used to prevent simultaneous logging from happening

### Create a logging object

ruffus.proxy\_logger.**setup\_std\_shared\_logger** (logger\_name, args)  
This function is a simple around wrapper around the python [logging](#) module.

This `logger_factory` example creates logging objects which can then be managed by proxy via `ruffus.proxy_logger.make_shared_logger_and_proxy()`

This can be:

- a disk log file
- a automatically backed-up (rotating) log.
- any log specified in a configuration file

These are specified in the `args` dictionary forwarded by `make_shared_logger_and_proxy()`

#### Parameters

- `logger_name` – name of log
- `args` – a dictionary of parameters forwarded from `make_shared_logger_and_proxy()`

Valid entries include:

- "level"**  
Sets the `threshold` for the logger.
- "config\_file"**  
The logging object is configured from this `configuration file`.
- "file\_name"**  
Sets disk log file name.
- "rotating"**  
Chooses a (rotating) log.
- "maxBytes"**  
Allows the file to rollover at a predetermined size
- "backupCount"**  
If `backupCount` is non-zero, the system will save old log files by appending the extensions `.1`, `.2`, `.3` etc., to the filename.
- "delay"**  
Defer file creation until the log is written to.
- "formatter"**  
Converts the message to a logged entry string. For example,

```
"%(asctime)s - %(name)s - %(levelname)s - %(message)s"
```



## OLD MANUAL: / TUTORIAL

### 5.1 Ruffus Manual: Table of Contents:

*manual\_introduction*

1. *follows*
2. *tasks\_as\_recipes*
3. *files*
4. *tasks\_and\_globs\_in\_inputs*
5. *tracing\_pipeline\_parameters*
6. *parallel\_processing*
7. *split*
8. *transform*
9. *merge*
10. *posttask*
11. *jobs\_limit*
12. *dependencies*
13. *onthefly*
14. *collate*
15. *advanced\_transform*
16. *parallel*
17. *check\_if\_upToDate*
18. *exceptions*
19. *logging*
20. *files\_re*

### 5.2 Ruffus Manual

The chapters of this manual go through each of the features of **Ruffus** in turn.

Some of these (especially those labelled **esoteric** or **deprecated**) may not be of interest to all users of **Ruffus**.

If you are looking for a quick introduction to **Ruffus**, you may want to look at the *Simple Tutorial* first, some of which content is shared with, or elaborated on, by this manual.

### 5.2.1 Introduction

The **Ruffus** module is a lightweight way to run computational pipelines.

Computational pipelines often become quite simple if we breakdown the process into simple stages.

---

**Note:** Ruffus refers to each stage of your pipeline as a *task*.

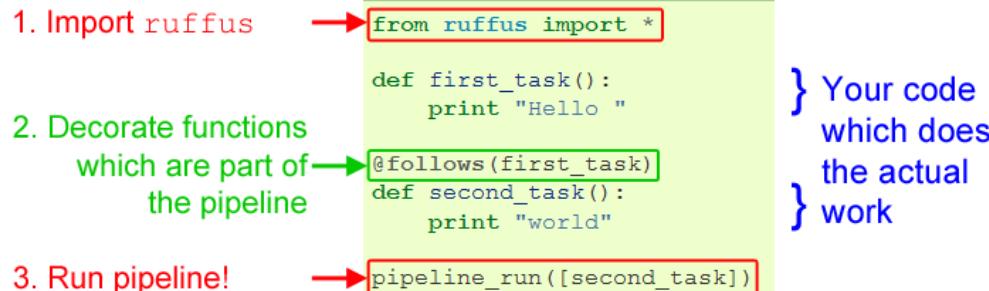
---

Let us start with the usual “Hello World”.

We have the following two python functions which we would like to turn into an automatic pipeline:

```
def first_task():
    print "Hello "
def second_task():
    print "world"
```

The simplest **Ruffus** pipeline would look like this:



The functions which do the actual work of each stage of the pipeline remain unchanged. The role of **Ruffus** is to make sure these functions are called in the right order, with the right parameters, running in parallel using multiprocessing if desired.

There are three simple parts to building a **ruffus** pipeline

1. importing ruffus
2. “Decorating” functions which are part of the pipeline
3. Running the pipeline!

### 5.2.2 Importing ruffus

The most convenient way to use ruffus is to import the various names directly:

```
from ruffus import *
```

This will allow **ruffus** terms to be used directly in your code. This is also the style we have adopted for this manual.

Category	Terms
<i>Pipeline functions</i>	pipeline_printout pipeline_printout_graph pipeline_run register_cleanup
<i>Decorators</i>	@follows @files @split @transform @merge @collate @posttask @jobs_limit @parallel @check_if_upToDate @files_re
<i>Loggers</i>	stderr_logger black_hole_logger
<i>Parameter disambiguating Indicators</i>	suffix regex inputs touch_file combine mkdir output_from

If any of these clash with names in your code, you can use qualified names instead:

```
import ruffus

ruffus.pipeline_printout("...")
```

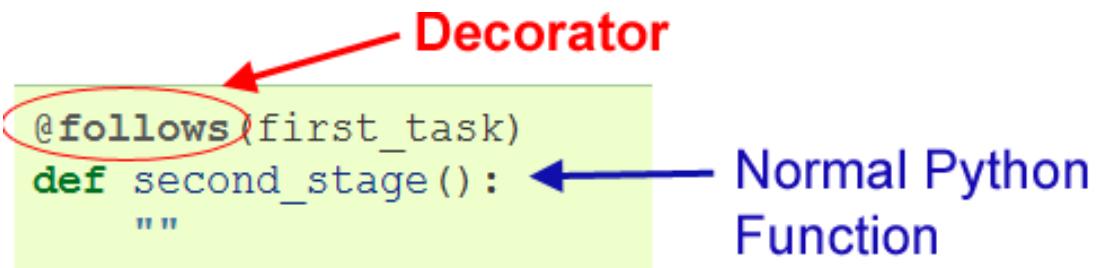
### 5.2.3 “Decorating” functions

You need to tag or *decorator* existing code to tell **Ruffus** that they are part of the pipeline.

---

**Note:** *decorators* are ways to tag or mark out functions.

They start with an @ prefix and take a number of parameters in parenthesis.



---

The **ruffus** decorator `@follows` makes sure that `second_task` follows `first_task`.

Multiple *decorators* can be used for each *task* function to add functionality to *Ruffus* pipeline functions.

However, the decorated python functions can still be called normally, outside of *Ruffus*.

*Ruffus decorators* can be added to (stacked on top of) any function in any order.

- More on `@follows` in `|manual.follows.chapter_num|`
- `@follows` syntax in detail

## 5.2.4 Running the pipeline

We run the pipeline by specifying the **last** stage (*task* function) of your pipeline. Ruffus will know what other functions this depends on, following the appropriate chain of dependencies automatically, making sure that the entire pipeline is up-to-date.

In our example above, because `second_task` depends on `first_task`, both functions are executed in order.

```
>>> pipeline_run([second_task], verbose = 1)
```

**Ruffus** by default prints out the verbose progress through your pipeline, interleaved with our Hello and World.

```
Start Task = first_task
Hello
    Job completed
Completed Task = first_task
Start Task = second_task
World
    Job completed
Completed Task = second_task
```

## 5.3 Chapter 1 : Arranging tasks into a pipeline with `@follows`

- Manual overview
- `@follows` syntax in detail

### 5.3.1 @follows

The order in which stages or *tasks* of a pipeline are arranged are set explicitly by the `@follows(...)` python decorator:

```
from ruffus import *
import sys

def first_task():
    print "First task"

@follows(first_task)
def second_task():
    print "Second task"

@follows(second_task)
def final_task():
    print "Final task"
```

the `@follows` decorator indicate that the `first_task` function precedes `second_task` in the pipeline.

---

**Note:** We shall see in *Chapter 2* that the order of pipeline *tasks* can also be inferred implicitly for the following decorators

- `@split(...)`
  - `@transform(...)`
  - `@merge(...)`
  - `@collate(...)`
- 

## Running

**Now we can run the pipeline by:**

```
pipeline_run([final_task])
```

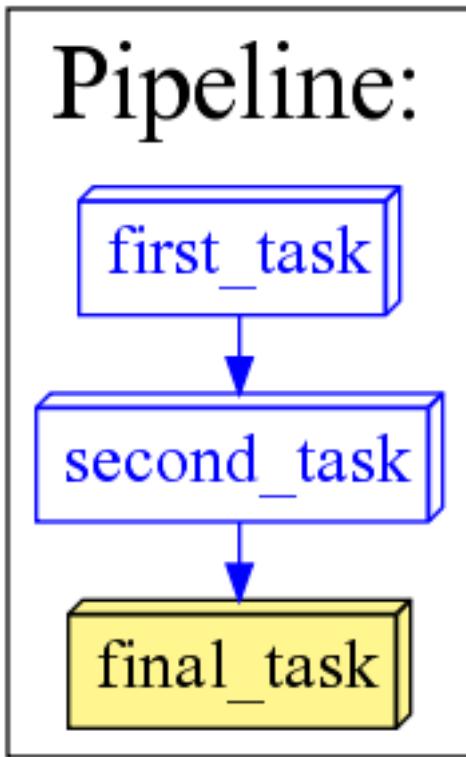
Because `final_task` depends on `second_task` which depends on `first_task`, all three functions will be executed in order.

## Displaying

**We can see a flowchart of our fledgling pipeline by executing:**

```
pipeline_printout_graph ( "manual_follows1.png",
                           "png",
                           [final_task],
                           no_key_legend=True)
```

producing the following flowchart



or in text format with:

```
pipeline_printout(sys.stdout, [final_task])
```

which produces the following:

```
Task = first_task
Task = second_task
Task = final_task
```

### 5.3.2 Defining pipeline tasks out of order

All this assumes that all your pipelined tasks are defined in order. (`first_task` before `second_task` before `final_task`)

This is usually the most sensible way to arrange your code.

If you wish to refer to tasks which are not yet defined, you can do so by quoting the function name as a string:

```
@follows("second_task")
def final_task():
    print "Final task"
```

You can refer to tasks (functions) in other modules, in which case the full qualified name must be used:

```
@follows("other_module.second_task")
def final_task():
    print "Final task"
```

### 5.3.3 Multiple dependencies

Each task can depend on more than one antecedent task.

This can be indicated either by stacking `@follows`:

```
@follows(first_task)
@follows("second_task")
def final_task():
    ""
```

or in a more concise way:

```
@follows(first_task, "second_task")
def final_task():
    ""
```

### 5.3.4 Making directories automatically with `mkdir`

A common prerequisite for any computational task, is making sure that the destination directories exist.

Ruffus provides special syntax to support this, using the special `mkdir` dependency. For example:

```
@follows(first_task, mkdir("output/results/here"))
def second_task():
    print "Second task"
```

will make sure that `output/results/here` exists before `second_task` is run.

In other words, it will make the `output/results/here` directory if it does not exist.

## 5.4 Chapter 2: *Tasks and Recipes*

- *Manual overview*

The python functions which do the actual work of each stage or *task* of a Ruffus pipeline are written by you.

The role of Ruffus is to make sure these functions are called in the right order, with the right parameters, running in parallel using multiprocessing if desired.

Ruffus manages the data flowing through your pipeline by supplying the correct parameters to your pipeline functions. In this way, you will get the following features for free:

1. only out-of-date parts of the pipeline will be re-run
2. multiple jobs can be run in parallel (on different processors if possible)
3. pipeline stages can be chained together automatically

Much of the functionality of **ruffus** involves determining the data flow through your pipeline, by governing how the output of one stage of the pipeline is supplied as parameters to the functions of the next.

### 5.4.1 Skip jobs which are up to date

Very often it will necessary to re-run a computational pipeline, because part of the data has changed. **Ruffus** will run only those stages of the pipeline which are absolutely necessary.

By default, **Ruffus** uses file modification times to see which parts of the pipeline are out of date, and which *tasks* need to be run again. This is so convenient that even if a pipeline is not file-based (if it, for example, uses database tables instead), it may be worth while to use dummy, “sentinel” files to manage the stages of a pipeline.

(It is also possible, as we shall see later, to add custom functions to determine which parts of the pipeline are out of date. see `@parallel` and `@check_if_upToDate`.)

### 5.4.2 *Inputs and Outputs* parameters

**Ruffus** treats the first two parameters of each job in each task as the *inputs* and *outputs* parameters respectively. If these parameters are strings, or are sequences which contain strings, these will be treated as the names of files required by and produced by that job. The presence and modification times of the *inputs* and *outputs* files will be used to check if it is necessary to rerun the job.

Apart from this, **Ruffus** imposes no other restrictions on the parameters for jobs, which are passed verbatim to task functions.

Most of the time, it is sensible to stick with file names (strings) in the *inputs* and *outputs* parameters but **Ruffus** does not try to second-guess what sort of data you will be passing through your pipelines (except that strings represent file names).

Thus, given the following over-elaborate parameters (parameter passing will be discussed in more detail from `\manual.files.chapter_num`):

```
[ [[1, 3], "afile.name", ("bfile.name", 72)],
  [[56, 3.3], set(custom_object(), "output.file")],
  33.3,
  "oops"]
```

This will be passed “as is” to your task function:

```
do_something([[1, 3], "afile.name", ("bfile.name", 72)],      # input
             [[56, 3.3], set(custom_object(), "output.file")],    # output
             33.3,                                                 # extra parameter
             "oops")                                              # extra parameter
```

**Ruffus** will interpret this as:

```
Input_parameter = [[1, 3], "afile.name", ("bfile.name", 72)]
Output_parameter = [[56, 3.3], set(custom_object(), "output.file")]
Other_parameter_1 = 33.3
Other_parameter_2 = "oops"
```

**Ruffus** disregards the *structure* of your data, only identifying the (nested) strings. Thus there are 2 input files:

```
"afile.name"
"bfile.name"
```

and 1 output file:

```
"output.file"
```

### 5.4.3 Checking if files are up to date

The following simple rules are used by **Ruffus**.

1. The pipeline stage will be rerun if:
  - If any of the *inputs* files are new (newer than the *output* files)
  - If any of the *output* files are missing
2. In addition, it is possible to run jobs which create files from scratch.
  - If no *inputs* file names are supplied, the job will only run if any *output* file is missing.
3. Finally, if no *outputs* file names are supplied, the job will always run.

The *example* in the next chapter shows how this works in practice.

### 5.4.4 Missing files

If the *inputs* files for a job are missing, the task function will have no way to produce its *output*. In this case, a `MissingInputFileError` exception will be raised automatically. For example,

```
task.MissingInputFileError: No way to run job: Input file ['a.1'] does not exist
for Job = ["a.1" -> "a.2", "A file"]
```

### 5.4.5 Caveats: Timestamp resolution

Note that modification times have precision to the nearest second under some older file systems (ext2/ext3?). This may be also be true for networked file systems.

**Ruffus** is very conservative, and assumes that files with *exactly* the same date stamp might have been created in the wrong order, and will treat the job as out-of-date. This would result in some jobs re-running unnecessarily, simple because an underlying coarse-grained file system does not distinguish between successively created files with sufficiently accuracy.

To get around this, **Ruffus** makes sure that each task is punctuated by a 1 second pause (via `time.sleep()`). If this gets in the way, and you are using a modern file system with nanosecond timestamp resolution, you can turn off the delay by setting `one_second_per_job` to `False` in `pipeline_run`

Later versions of **Ruffus** will allow file modification times to be saved at higher precision in a log file or database to get around this.

## 5.5 Chapter 3: *Passing parameters to the pipeline with @files*

- *Manual overview*
- *@files syntax in detail*

The python functions which do the actual work of each stage or *task* of a **Ruffus** pipeline are written by you.

The role of **Ruffus** is to make sure these functions are called in the right order, with the right parameters, running in parallel using multiprocessing if desired.

The easiest way to specify parameters to *Ruffus task* functions is to use the `@files` decorator.

### 5.5.1 @files

Running this code:

```
from ruffus import *

@files('a.1', ['a.2', 'b.2'], 'A file')
def single_job_io_task(infile, outfiles, text):
    for o in outfiles: open(o, "w")

# prepare input file
open('a.1', "w")

pipeline_run()
```

Is equivalent to calling:

```
single_job_io_task('a.1', ['a.2', 'b.2'], 'A file')
```

And produces:

```
>>> pipeline_run()
Job = [a.1 -> [a.2, b.2], A file] completed
Completed Task = single_job_io_task
```

**Ruffus** will automatically check if your task is up to date. The second time `pipeline_run()` is called, nothing will happen. But if you update `a.1`, the task will rerun:

```
>>> open('a.1', "w")
>>> pipeline_run()
Job = [a.1 -> [a.2, b.2], A file] completed
Completed Task = single_job_io_task
```

See *chapter 2* for a more in-depth discussion of how **Ruffus** decides which parts of the pipeline are complete and up-to-date.

### 5.5.2 Running the same code on different parameters in parallel

Your pipeline may require the same function to be called multiple times on independent parameters. In which case, you can supply all the parameters to `@files`, each will be sent to separate jobs that may run in parallel if necessary. **Ruffus** will check if each separate *job* is up-to-date using the *inputs* and *outputs* (first two) parameters (See the *chapter 2* ).

For example, if a sequence (e.g. a list or tuple) of 5 parameters are passed to `@files`, that indicates there will also be 5 separate jobs:

```
from ruffus import *
parameters = [
    [ 'job1.file'                 ],          # 1st job
    [ 'job2.file', 4               ],          # 2st job
    [ 'job3.file', [3, 2] ]           ,          # 3st job
```

```
[ 67, [13, 'job4.file'] ], # 4st job
[ 'job5.file' ], # 5st job
]
@files(parameters)
def task_file(*params):
    """
```

**Ruffus** creates as many jobs as there are elements in `parameters`.

In turn, each of these elements consist of series of parameters which will be passed to each separate job.

Thus the above code is equivalent to calling:

```
task_file('job1.file')
task_file('job2.file', 4)
task_file('job3.file', [3, 2])
task_file(67, [13, 'job4.file'])
task_file('job5.file')
```

What `task_file()` does with these parameters is up to you!

The only constraint on the parameters is that **Ruffus** will treat any first parameter of each job as the *inputs* and any second as the *output*. Any strings in the *inputs* or *output* parameters (including those nested in sequences) will be treated as file names.

Thus, to pick the parameters out of one of the above jobs:

```
task_file(67, [13, 'job4.file'])
```

```
inputs == 67
outputs == [13, 'job4.file']
```

The solitary output filename is `job4.file`

## Checking if jobs are up to date

Usually we do not want to run all the stages in a pipeline but only where the input data has changed or is no longer up to date.

One easy way to do this is to check the modification times for files produced at each stage of the pipeline.

Let us first create our starting files `a.1` and `b.1`

We can then run the following pipeline function to create

- `a.2` from `a.1` and
- `b.2` from `b.1`

```
# create starting files
open("a.1", "w")
open("b.1", "w")
```

```
from ruffus import *
parameters = [
    [ 'a.1', 'a.2', 'A file'], # 1st job
    [ 'b.1', 'b.2', 'B file'], # 2nd job
]

@files(parameters)
def parallel_io_task(infile, outfile, text):
    # copy infile contents to outfile
    infile_text = open(infile).read()
    f = open(outfile, "w").write(infile_text + "\n" + text)

pipeline_run()
```

This produces the following output:

```
>>> pipeline_run()
Job = [a.1 -> a.2, A file] completed
Job = [b.1 -> b.2, B file] completed
Completed Task = parallel_io_task
```

If you called `pipeline_run()` again, nothing would happen because the files are up to date:

a.2 is more recent than a.1 and  
b.2 is more recent than b.1

However, if you subsequently modified a.1 again:

```
open("a.1", "w")
pipeline_run(verbose = 1)
```

you would see the following:

```
>>> pipeline_run([parallel_io_task])
Task = parallel_io_task
Job = ["a.1" -> "a.2", "A file"] completed
Job = ["b.1" -> "b.2", "B file"] unnecessary: already up to date
Completed Task = parallel_io_task
```

The 2nd job is up to date and will be skipped.

## 5.6 Chapter 4: Chaining pipeline Tasks together automatically

- *Manual overview*

In the previous chapter, we explained that **ruffus** determines the data flow through your pipeline by calling your *task* functions (normal python functions written by you) with the right parameters at the right time, making sure that

1. only out-of-date parts of the pipeline will be re-run
2. multiple jobs can be run in parallel (on different processors if possible)
3. pipeline stages can be chained together automatically

This chapter is devoted to the last item: how the output of one stage of the pipeline is piped into as the input of the next stage.

### 5.6.1 Tasks in the *inputs* parameters: Implicit dependencies

**Ruffus** treats the first two parameters of each job in each task as the *inputs* and *outputs* parameters respectively. If the *inputs* parameter contains strings, these will be treated as the names of files required by that job.

If the *inputs* parameter contains any *tasks*, **Ruffus** will take the output from these specified tasks as part of the current *inputs* parameter. In addition, such tasks will be listed as prerequisites, much as if you had included them in a separate @follows decorator.

For example, supposed we wanted to take the output files from `task1` and feed them automatically to `task2`, we might write the following code

```
task1_ouput_files = ("task1.output_a", "task1.output_b", "task1.output_c")

@follows(task1)
@files(task1_ouput_files, "task2.output")
def task2(input, output):
    pass
```

This can be replaced by the much more concise syntax:

```
@files(task1, "task2.output")
def task2(input, output):
    pass
```

**This means:**

- Take the output from `task1`, and feed it automatically into `task2`.
- Also make sure that `task2` becomes a dependency of `task1`.

In other words, `task1` and `task2` have been chained together automatically. This is both a great convenience and makes the flow of data through a pipeline much clearer.

### 5.6.2 Referring to tasks by name in the *inputs* parameters

*Chapter 1* explains that task functions can be defined in any order so long as undefined tasks are referred to by their (fully qualified if necessary) function name string.

You can similarly refer to tasks in the *inputs* parameter by name, as a text string. Normally **Ruffus** assumes that strings are file names. To indicate that that you are referring to task function names instead, you need to wrap the relevant parameter or (nested) parts of the parameter with the indicator object `output_from("task_name")`. Thus,

```
@split(["a.file", ("b.file", output_from("task1", 76, "task2"))], "*.*.split")
def task2(input, output):
    pass
```

is equivalent to:

```
@split(["a.file", ("b.file", (task1, 76, task2))], "*.*.split")
def task2(input, output):
    pass
```

### 5.6.3 Globs in the *inputs* parameters

As a syntactic convenience, **Ruffus** also allows you to specify a *glob* pattern (e.g. `*.txt`) in the *input* parameter, it will be expanded automatically to the actually matching file names. This applies to any strings within *inputs* which contain the letters: `* ? [ ]`.

### 5.6.4 Mixing globbs, tasks and files as inputs

**Ruffus** is very flexible in allowing you to mix *glob* patterns, references to tasks and file names in the data structures you pass as the **inputs** parameters.

Suppose, in the previous example,

- that `task1` produces the files

```
"task1.output_a"  
"task1.output_b"  
"task1.output_c"
```

- that the following additional files are also present

```
"extra.a"  
"extra.c"
```

Then,

```
@files(["1_more.file", "2_more.file", task1, "extra.*"], "task2.output")  
def task2(input, output):  
    pass
```

would result in the combination of the specified file name, the expansion of the *glob*, and the results from the previous task:

```
input == [  
    "1_more.file" , # specified file  
    "2_more.file" , # specified file  
    "task1.output_a", # from previous task  
    "task1.output_b", # from previous task  
    "task1.output_c", # from previous task  
    "extra.a" , # from glob expansion  
    "extra.c" , # from glob expansion  
]
```

In other words, *glob* patterns and tasks are expanded “in place” when they are part of python lists, sets, or tuples.

### 5.6.5 Appending globbs or tasks to pre-existing lists, sets or tuples

Sometimes we want to the *inputs* parameter to contain be a combination of *globbs* and tasks, and an existing list of file names.

To elaborate on the above example, suppose we have a list of files:

```
file_list = [ "1_more.file",  
              "2_more.file" ]
```

Now we want the input to `task2` to be:

```
file_list + task1 + "extra.*"
```

The closest that we can express this in python syntax is by turning task1 and the *glob* to a list first then adding them together:

```
@files(file_list + [task1] + ["extra.*"], "task2.output")
def task2(input, output):
    pass
```

The same also works with tuples:

```
file_list = ("1_more.file",
             "2_more.file")

@files(file_list + (task1, "extra.*"), "task2.output")
def task2(input, output):
    pass
```

and sets (using the set concatenation operator):

```
file_list = set([
    "1_more.file",
    "2_more.file"])

@files(file_list | set([task1 + "extra.*"]), "task2.output")
def task2(input, output):
    pass
```

## 5.6.6 Understanding complex *inputs* and *outputs* parameters

In all cases, **Ruffus** tries to do the right thing, and to make the simple or obvious case require the simplest, least onerous syntax.

If sometimes **Ruffus** does not behave the way you expect, please write to the authors: it may be a bug!

In all other cases, the best thing to do, is write your **Ruffus** specifications, and check the results of *pipeline\_printout* to make sure that your wishes are properly reflected in the parameters sent to your pipelined tasks.

In other words, read the *next chapter!*

## 5.7 Chapter 5: Tracing pipeline parameters

- *Manual overview*

The trickiest part of developing pipelines is understanding how your data flows through the pipeline.

In **Ruffus**, your data is passed from one task function to another down the pipeline by the chain of linked parameters. Sometimes, it may be difficult to choose the right **Ruffus** syntax at first, or to understand which parameters in what format are being passed to your function.

Whether you are learning how to use **ruffus**, or trying out a new feature in **ruffus**, or just have a horrendously complicated pipeline to debug (we have colleagues with >100 criss-crossing pipelined stages), your best friend is *pipeline\_printout*(...).

**pipeline\_printout** displays the parameters which would be passed to each task function for each job in your pipeline. In other words, it traces how each of the functions in the pipeline are called in detail.

It makes good sense to alternate between calls to **pipeline\_printout** and **pipeline\_run** in the development of **Ruffus** pipelines (perhaps with the use of a command-line option), so that you always know exactly how the pipeline is being invoked.

### 5.7.1 Printing out which jobs will be run

**pipeline\_printout** is called in exactly the same way as **pipeline\_run** but instead of running the pipeline, just prints the tasks which are and are not up-to-date.

The `verbose` parameter controls how much detail is displayed.

```
verbose = 0 : prints nothing
verbose = 1 : logs warnings and tasks which are not up-to-date and which will be run
verbose = 2 : logs doc strings for task functions as well
verbose = 3 : logs job parameters for jobs which are out-of-date
verbose = 4 : logs list of up-to-date tasks but parameters for out-of-date jobs
verbose = 5 : logs parameters for all jobs whether up-to-date or not
verbose = 10: logs messages useful only for debugging ruffus pipeline code
```

Let us take the two step *pipeline* from the tutorial. *Pipeline\_printout(...)* by default merely lists the two tasks which will be run in the pipeline:

```
>>> pipeline_printout(sys.stdout, [second_task])
Tasks which will be run:
Task = first_task
Task = second_task } Tasks which will be run
```

To see the input and output parameters of out-of-date jobs in the pipeline, we can increase the verbosity from the default (1) to 3:

```
>>> pipeline_printout(sys.stdout, [second_task], verbose = 3 )
```

Tasks which will be run:

```
Task = first_task
    Job = [None
           ->job1.stage1] ← Job parameters
        Job needs update: Missing file job1.stage1
    Job = [None
           ->job2.stage1] ← Job parameters
        Job needs update: Missing file job2.stage1

Task = second_task
    Job = [job1.stage1
           ->job1.stage2,      1st_job] ← Job parameters
        Job needs update: Missing file job1.stage1
    Job = [job2.stage1
           ->job2.stage2,      2nd job] ← Job parameters
        Job needs update: Missing file job2.stage1
```

This is very useful for checking that the input and output parameters have been specified correctly.

### 5.7.2 Determining which jobs are out-of-date or not

It is often useful to see which tasks are or are not up-to-date. For example, if we were to run the pipeline in full, and then modify one of the intermediate files, the pipeline would be partially out of date.

Let us start by run the pipeline in full but then modify `job1.stage1` so that the second task is no longer up-to-date:

```
pipeline_run([second_task])

# modify job1.stage1
open("job1.stage1", "w").close()
```

At a verbosity of 5, even jobs which are up-to-date will be displayed. We can now see that the there is only one job in `second_task(...)` which needs to be re-run because `job1.stage1` has been modified after `job1.stage2` (highlighted in blue):

```
>>> pipeline_printout(sys.stdout, [second_task], verbose = 5)

Tasks which are up-to-date:
Task = first_task
    Job = [None
           ->job1.stage1]
        Job up-to-date
    Job = [None
           ->job2.stage1]
        Job up-to-date

Tasks which will be run:
Task = second_task
    Job = [job1.stage1
           ->job1.stage2,      1st_job]
        Job needs update:
            Need update file times= [
                [(1269025787.0, 'job1.stage1')],
                [(1269025785.0, 'job1.stage2')] ]
    Job = [job2.stage1
           ->job2.stage2,      2nd_job]
        Job up-to-date
```

Out of date job with the  
file times of the relevant  
input/output files displayed

---

## 5.8 Chapter 6: *Running Tasks and Jobs in parallel*

- *Manual overview*

### 5.8.1 Multi Processing

*Ruffus* uses python `multiprocessing` to run each job in a separate process.

This means that jobs do *not* necessarily complete in the order of the defined parameters. Task hierachies are, of course, inviolate: upstream tasks run before downstream, dependent tasks.

Tasks that are independent (i.e. do not precede each other) may be run in parallel as well.

The number of concurrent jobs can be set in `pipeline_run`:

```
pipeline_run([parallel_task], multiprocess = 5)
```

If `multiprocess` is set to 1, then jobs will be run on a single process.

### 5.8.2 Data sharing

Running jobs in separate processes allows *Ruffus* to make full use of the multiple processors in modern computers. However, some of the `multiprocessing` guidelines should be borne in mind when writing *Ruffus* pipelines. In particular:

- Try not to pass large amounts of data between jobs, or at least be aware that this has to be marshalled across process boundaries.

- Only data which can be `pickled` can be passed as parameters to *Ruffus* task functions. Happily, that applies to almost any Python data type. The use of the rare, unpicklable object will cause python to complain (fail) loudly when *Ruffus* pipelines are run.

## 5.9 Chapter 7: *Splitting up large tasks / files with @split*

- *Manual overview*
- `@split` syntax in detail

A common requirement in computational pipelines is to split up a large task into small jobs which can be run on different processors, (or sent to a computational cluster). Very often, the number of jobs depends dynamically on the size of the task, and cannot be known for sure beforehand.

*Ruffus* uses the `@split` decorator to indicate that the *task* function will produce an indeterminate number of output files.

### 5.9.1 `@split`

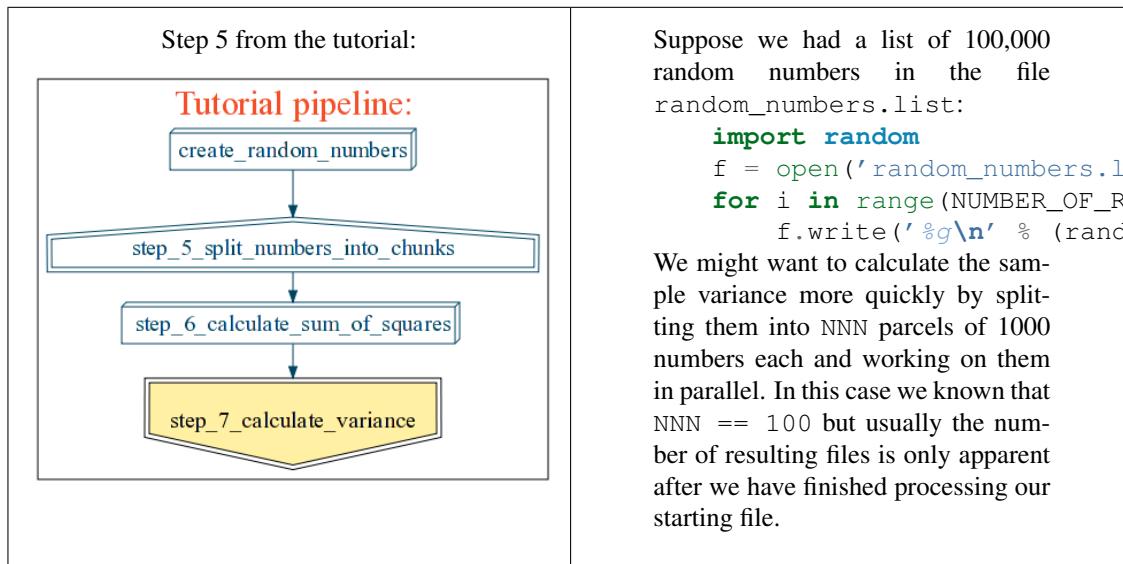
This example is borrowed from *step 4* of the simple tutorial.

---

**Note:** See accompanying *Python Code*

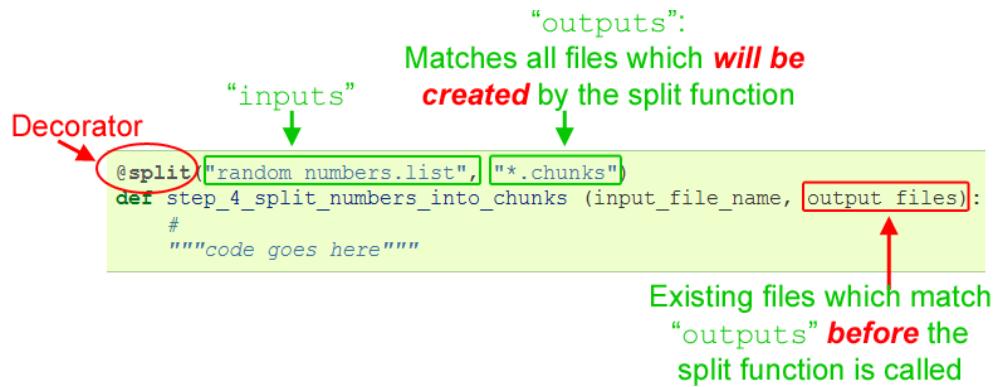
---

#### Splitting up a long list of random numbers to calculate their variance



Our pipeline function needs to take the random numbers file `random_numbers.list`, read the random numbers from it, and write to a new file every 100 lines.

The *Ruffus* decorator `@split` is designed specifically for splitting up *inputs* into an indeterminate NNN number of *outputs*:



Ruffus will set

```
input_file_name to "random_numbers.list"
output_files to all files which match *.chunks (i.e. "1.chunks", "2.chunks" etc.).
```

## 5.9.2 Output files

The *output* (second) parameter of **@split** usually contains a *glob* pattern like the `*.chunks` above.

---

**Note:** Ruffus is quite relaxed about the contents of the *output* parameter. Strings are treated as file names. Strings containing *glob* pattern are expanded. Other types are passed verbatim to the decorated task function.

---

The files which match the *glob* will be passed as the actual parameters to the job function. Thus, the first time you run the example code `*.chunks` will return an empty list because no `.chunks` files have been created, resulting in the following:

```
step_4_split_numbers_into_chunks ("random_numbers.list", [])
```

After that `*.chunks` will match the list of current `.chunks` files created by the previous pipeline run.

File names in *output* are generally out of date or superfluous. They are useful mainly for cleaning-up detritus from previous runs (have a look at `step_4_split_numbers_into_chunks(...)`).

---

**Note:** It is important, nevertheless, to specify correctly the list of *output* files. Otherwise, dependent tasks will not know what files you have created, and it will not be possible automatically to chain together the *output* of this pipeline task into the *inputs* of the next step.

You can specify multiple *glob* patterns to match *all* the files which are the result of the splitting task function. These can even cover different directories, or groups of file names. This is a more extreme example:

```
@split("input.file", ['a*.bits', 'b*.pieces', 'somewhere_else/c*.stuff'])
def split_function (input_filename, output_files):
    "Code to split up 'input.file'"
```

---

The actual resulting files of this task function are not constrained by the file names in the *output* parameter of the function. The whole point of **@split** is that number of resulting output files cannot be known beforehand, after all.

## Example

Suppose `random_numbers.list` can be split into four pieces, this function will create `1.chunks`, `2.chunks`, `3.chunks`, `4.chunks`

Subsequently, we receive a larger `random_numbers.list` which should be split into 10 pieces. If the pipeline is called again, the task function receives the following parameters:

```
step_4_split_numbers_into_chunks("random_numbers.list",
["1.chunks",          "# previously created files
 "2.chunks",          "# 
 "3.chunks",          "# 
 "4.chunks" ])        "#
```

This doesn't stop the function from creating the extra `5.chunks`, `6.chunks` etc.

---

**Note:** Any tasks `@following` and specifying `step_4_split_numbers_into_chunks(...)` as its `inputs` parameter is going to receive `1.chunks`, ..., `10.chunks` and not merely the first four files.

In other words, dependent / down-stream tasks which obtain output files automatically from the task decorated by `@split` receive the most current file list. The `glob` patterns will be matched again to see exactly what files the task function has created in reality *after* the task completes.

---

## 5.10 Chapter 8: Applying the same recipe to create many different files with `@transform`

- *Manual overview*
- `@transform` syntax in detail

Sometimes you might have a list of data files which you might want to send to the same pipelined function, to apply the same operation. The best way to manage this would be to produce a corresponding list of results files:

Compiling c source files might `@transform` an `a.c` file to an `a.o` file.

A grep operation might `@transform` a `plays.king_lear.txt` file to an `plays.king_lear.counts` file.

*Ruffus* uses the `@transform` decorator for this purpose.

When you `@transform` your data from one file type to another, you are not restricted just to changing the file suffix. We shall see how, with the full power of regular expressions behind you, you can sort the resulting data into different directories, add indices and so on.

### 5.10.1 `@transform`

#### Worked example: calculating sums and sum of squares in parallel

This example is borrowed from *step 5* of the simple tutorial.

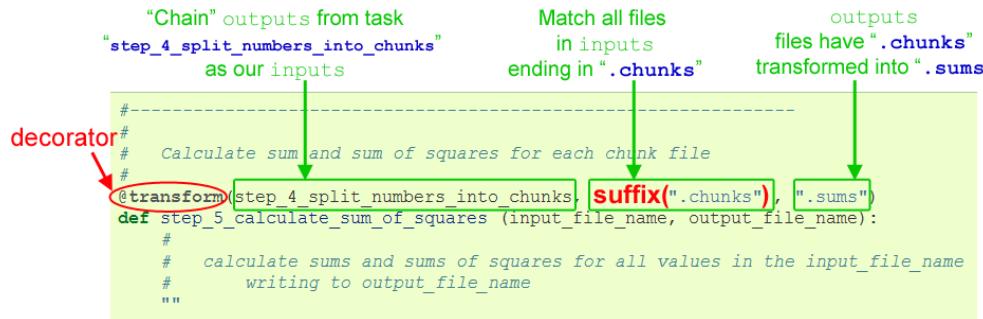
---

**Note:** See *example code here*

---

Given a set of files, each with a set of random numbers, we want to calculate their sums and sum of squares. The easiest way to do this is by providing a recipe for transforming a `*.chunk` file containing a list of numbers into a `*.sums` file with our sums and sum of squares.

*Ruffus* magically takes care of applying the same recipe (task function) to all the different data files in parallel.



The `@transform` decorator tells *Ruffus* to take files from the step 4 task (i.e. `*.chunks`), and produce files having the `.sums` suffix instead. ending.

Thus if `step_4_split_numbers_into_chunks` created

```
"1.chunks"
"2.chunks"
"3.chunks"
```

This would result in the following function calls:

```
step_5_calculate_sum_of_squares ("1.chunk", "1.sum")
step_5_calculate_sum_of_squares ("2.chunk", "2.sum")
step_5_calculate_sum_of_squares ("3.chunk", "3.sum")

# etc...
```

## 5.10.2 Using `suffix(...)` to change give each output file a new suffix

The `suffix` specification indicates that

- only filenames with ending with the suffix term (e.g. `.chunk`) should be considered
- The text matching the suffix term should be replaced with the string in the output pattern.

This example assumes that both the *inputs* and the *outputs* consist each of a single string but **Ruffus** places no such constraints on the data flowing through your pipeline.

- If there are multiple file names (strings) contained within each *inputs* parameter, then only the first will be used to generate the *output*
- Each string that is encountered in each *output* parameter will be used for suffix replacement.

### An example with more complex data structures

This will become much clearer with this example:

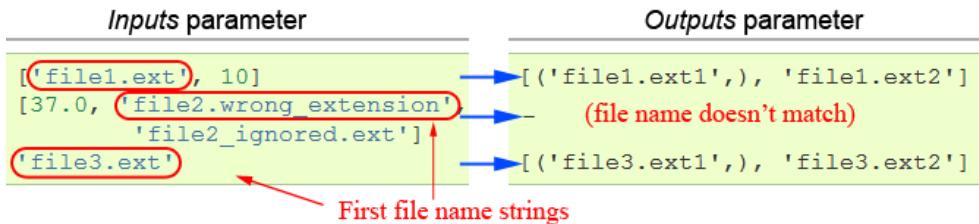
```
inputs = [
    ["file1.ext", 10      ],                      # job 1
    [37.0, "file2.wrong_extension",
     "file2_ignored.ext"],                      # job 2
    "file3.ext"                                    # job 3
]
```

```
@transform(inputs, suffix(".ext"), [(".ext1", ), ".ext2"])
def pipelinetask (input_file_name, output_file_name):
    ""
```

Granted, it may seem rather odd that the *inputs* parameter including numbers as well as file names, but **Ruffus** does not second guess how you wish to arrange your pipelines.

*inputs* contains the parameters for three jobs.

In each case, the first file name string encountered will be used to generate the *output* parameter:



**Note:** The first filename in the prospective job #2 does not have the .ext suffix so this job will be eliminated.

Thus, the original code:

```
@transform(inputs, suffix(".ext"), [(15, ".ext1"), ".ext2"])
def pipelinetask (input_file_name, output_file_name):
    ""
```

is equivalent to calling:

```
pipelinetask(["file1.ext", 10], [(15, 'file1.ext1'), 'file1.ext2']) # job 1
pipelinetask("file3.ext", [(15, 'file3.ext1'), 'file3.ext2']) # job 3
```

Hopefully, your code will simpler than this rather pathological case!

### 5.10.3 Regular expressions `regex(...)` provide maximum flexibility

Exactly the same function could be written using regular expressions:

```
@transform(inputs, regex(".ext"), [(15, ".ext1"), ".ext2"])
def pipelinetask (input_file_name, output_file_name):
    ""
```

However, regular expressions are not limited to suffix matches.

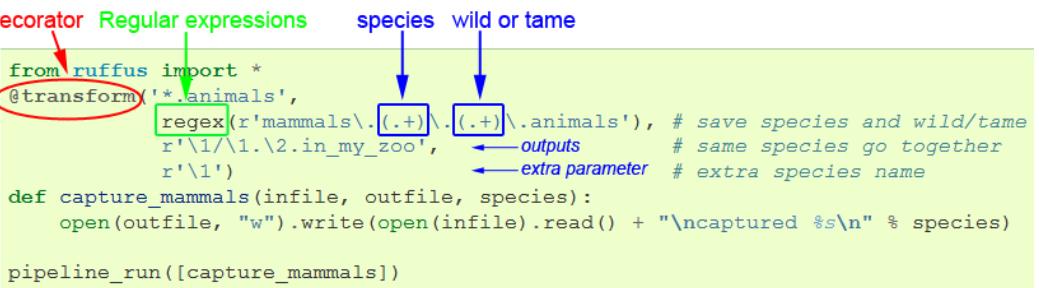
We can sort our *outputs* to different subdirectories, depending on category.

Our example starts off with data file for different zoo animals.

We are only interested in mammals, and we would like the files of each species to end up in its own directory after processing.

Starting with these species files:

```
"mammals.tiger.wild.animals"
"mammals.lion.wild.animals"
"mammals.lion.handreared.animals"
"mammals.dog.tame.animals"
"mammals.dog.wild.animals"
"reptiles.crocodile.wild.animals"
```



The diagram shows a code snippet with annotations:

- Decorator**: Points to the `@transform` decorator.
- Regular expressions**: Points to the regular expression pattern `r'mammals\.(.+)\.(.+)\.animals'`.
- species**: Points to the variable `species` in the `def capture_mammals` function.
- wild or tame**: Points to the variable `wild or tame` in the `def capture_mammals` function.
- outputs**: Points to the output files generated by the pipeline run.
- extra parameter**: Points to the `species` parameter in the `capture_mammals` function.
- extra species name**: Points to the `species` parameter in the `capture_mammals` function.

```
from ruffus import *
@transform '*.animals',
    regex(r'mammals\.(.+)\.(.+)\.animals'), # save species and wild/tame
    r'\1/\1.\2.in_my_zoo',                  ← outputs      # same species go together
    r'\1')                                  ← extra parameter # extra species name
def capture_mammals(infile, outfile, species):
    open(outfile, "w").write(open(infile).read() + "\ncaptured %s\n" % species)
pipeline_run([capture_mammals])
```

Then, the following:

will put each captured mammal in its own directory:

```
>>> pipeline_run([capture_mammals])
Job = [mammals.dog.tame.animals          -> dog/dog.tame.in_my_zoo, dog] completed
Job = [mammals.dog.wild.animals           -> dog/dog.wild.in_my_zoo, dog] completed
Job = [mammals.lion.handreared.animals   -> lion/lion.handreared.in_my_zoo, lion] completed
Job = [mammals.lion.wild.animals          -> lion/lion.wild.in_my_zoo, lion] completed
Job = [mammals.tiger.wild.animals         -> tiger/tiger.wild.in_my_zoo, tiger] completed
Completed Task = capture_mammals
```

---

**Note:** The code can be found [here](#)

---

## 5.11 Chapter 9: Merge *multiple input into a single result*

- *Manual overview*
- `@merge` syntax in detail

At the conclusion of our pipeline, or at key selected points, we might need a summary of our progress, gathering data from a multitude of files or disparate *inputs*, and summarised in the *output* of a single *job*.

Ruffus uses the `@merge` decorator for this purpose.

Although, `@merge` tasks multiple *inputs* and produces a single *output*, **Ruffus** is again agnostic as to the sort of data contained within *output*. It can be a single (string) file name, or an arbitrary complicated nested structure with numbers, objects etc. As always, strings contained (even with nested sequences) within *output* will be treated as file names for the purpose of checking if the *task* is up-to-date.

### 5.11.1 `@merge`

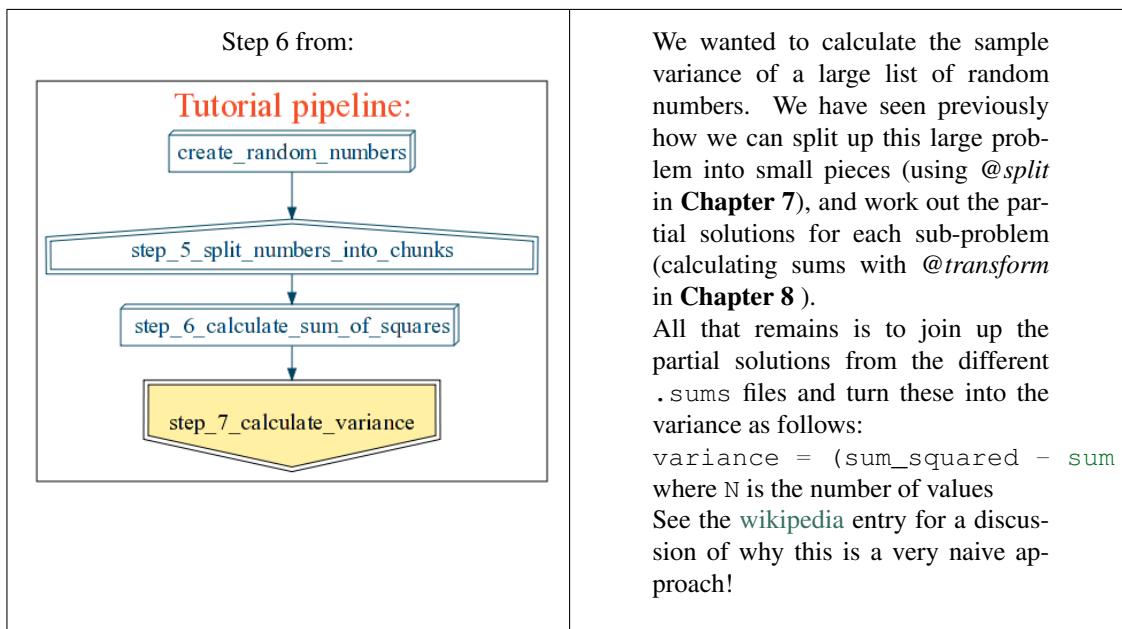
This example is borrowed from *step 6* of the simple tutorial.

---

**Note:** *Accompanying Python Code*

---

## Combining partial solutions: Calculating variances



To do this, all we have to do is go through all the values in `*.sums`, i.e. add up the sums and `sum_squared` for each chunk. We can then apply the above (naive) formula.

### Merging files is straightforward in Ruffus:

```

@merge(step_5_calculate_sum_of_squares, "variance.result")
def step_6_calculate_variance (input_file_names, output_file_name):
    #
    #   add together sums and sums of squares from each input_file_name
    #   calculate variance and write to output_file_name
    """
  
```

The `@merge` decorator tells *Ruffus* to take all the files from the step 5 task (i.e. `*.sums`), and produce a merge file in the form of `variance.result`.

Thus if `step_5_calculate_sum_of_squares` created

1. `sums` and
2. `sums` etc.

This would result in the following function call:

```
step_6_calculate_variance (["1.sum", "2.sum"], "variance.result")
```

The final result is, of course, in `variance.result`.

## 5.12 Chapter 10: *Signal the completion of each stage of our pipeline with @posttask*

- *Manual overview*
- `@posttask` syntax in detail

It is often useful to signal the completion of each task by specifying a specific action to be taken or function to be called. This can range from printing out some message, or touching some sentinel file,

to emailing the author. This is particular useful if the *task* is a recipe apply to an unspecified number of parameters in parallel in different *jobs*. If the task is never run, or if it fails, needless-to-say no task completion action will happen.

*Ruffus* uses the `@posttask` decorator for this purpose.

### 5.12.1 `@posttask`

We can signal the completion of each task by specifying one or more function(s) using `@posttask`

```
from ruffus import *

def task_finished():
    print "hooray"

@posttask(task_finished)
@files(None, "a.1")
def create_if_necessary(input_file, output_file):
    open(output_file, "w")

pipeline_run([create_if_necessary])
```

This is such a short function, we might as well write it in-line:

```
@posttask(lambda: sys.stdout.write("hooray\n"))
@files(None, "a.1")
def create_if_necessary(input_file, output_file):
    open(output_file, "w")
```

---

**Note:** The function(s) provided to `@posttask` will be called if the pipeline passes through a task, even if none of its jobs are run because they are up-to-date. This happens when an upstream task is out-of-date, and the execution passes through this point in the pipeline. See the example in *Chapter 9* of this manual.

---

### 5.12.2 `touch_file`

The most common way to note the completion of a task is to create some sort of “flag” file. Each stage in a traditional make pipeline would contain a `touch completed.flag`.

This is so common that **Ruffus** provides a special shorthand called `touch_file`:

```
from ruffus import *

@posttask(touch_file("task_completed.flag"))
@files(None, "a.1")
def create_if_necessary(input_file, output_file):
    open(output_file, "w")

pipeline_run([create_if_necessary])
```

### 5.12.3 Adding several post task actions

You can, of course, add more than one different action to be taken on completion of the task, either by stacking up as many `@posttask` decorators as necessary, or by including several functions in the same `@posttask`:

```
@posttask(print_hooray, print_whoppee)
@posttask(print_hip_hip, touch_file("sentinel_flag"))
@files(None, "a.1")
def your_pipeline_function (input_file_names, output_file_name):
    ""
```

## 5.13 Chapter 11: *Manage concurrency for a specific task with @jobs\_limit*

- *Manual overview*
- *@jobs\_limit* syntax in detail

### 5.13.1 @jobs\_limit

Calling `pipeline_run(multiprocess = NNN)` allows multiple jobs (from multiple independent tasks) to be run in parallel. However, there are some operations which consume so many resources that we might want them to run with less or no concurrency.

For example, we might want to download some files via FTP but the server restricts requests from each IP address. Even if the rest of the pipeline is running 100 jobs in parallel, the FTP downloading must be restricted to 2 files at a time. We would really like to keep the pipeline running as is, but let this one operation run either serially, or with little concurrency.

If `setting multiprocess = NNN` sets the pipeline-wide concurrency to NNN, then `@jobs_limit(MMM)` sets concurrency at a much finer level, at MMM just for jobs in the indicated task.

The optional name (e.g. `@jobs_limit(3, "ftp_download_limit")`) allows the same limit to be shared across multiple tasks. To be pedantic: a limit of 3 jobs at a time would be applied across all tasks which have a `@jobs_limit` named "ftp\_download\_limit":

```
from ruffus import *

# make list of 10 files
@split(None, "*stage1")
def make_files(input_file, output_files):
    for i in range(10):
        if i < 5:
            open("%d.small_stage1" % i, "w")
        else:
            open("%d.big_stage1" % i, "w")

@jobs_limit(3, "ftp_download_limit")
@transform(make_files, suffix(".small_stage1"), ".stage2")
def stage1_small(input_file, output_file):
    open(output_file, "w")

@jobs_limit(3, "ftp_download_limit")
@transform(make_files, suffix(".big_stage1"), ".stage2")
def stage1_big(input_file, output_file):
    open(output_file, "w")

@jobs_limit(5)
@transform([stage1_small, stage1_big], suffix(".stage2"), ".stage3")
def stage2(input_file, output_file):
```

```
open(output_file, "w")  
  
pipeline_run([stage2], multiprocess = 10)
```

will run the 10 jobs of stage1\_big and stage1\_small 3 at a time (highlighted in blue), a limit shared across the two tasks. stage2 jobs run 5 at a time (in red). These limits override the numbers set in pipeline\_run (multiprocess = 10):

```
>>> pipeline_run([stage2], multiprocess = 10)  
Job = [None -> *stage1] completed  
Completed Task = make_files  
    Job = [5.big_stage1 -> 5.stage2] completed  
    Job = [6.big_stage1 -> 6.stage2] completed  
    Job = [7.big_stage1 -> 7.stage2] completed  
  
    Job = [8.big_stage1 -> 8.stage2] completed  
    Job = [9.big_stage1 -> 9.stage2] completed  
Completed Task = stage1_big  
    Job = [0.small_stage1 -> 0.stage2] completed  
  
    Job = [1.small_stage1 -> 1.stage2] completed  
    Job = [2.small_stage1 -> 2.stage2] completed  
    Job = [3.small_stage1 -> 3.stage2] completed  
  
    Job = [4.small_stage1 -> 4.stage2] completed  
  
Completed Task = stage1_small  
    Job = [0.stage2 -> 0.stage3] completed  
    Job = [1.stage2 -> 1.stage3] completed  
    Job = [3.stage2 -> 3.stage3] completed  
    Job = [2.stage2 -> 2.stage3] completed  
    Job = [5.stage2 -> 5.stage3] completed  
  
    Job = [4.stage2 -> 4.stage3] completed  
    Job = [6.stage2 -> 6.stage3] completed  
    Job = [9.stage2 -> 9.stage3] completed  
    Job = [7.stage2 -> 7.stage3] completed  
    Job = [8.stage2 -> 8.stage3] completed  
  
Completed Task = stage2
```

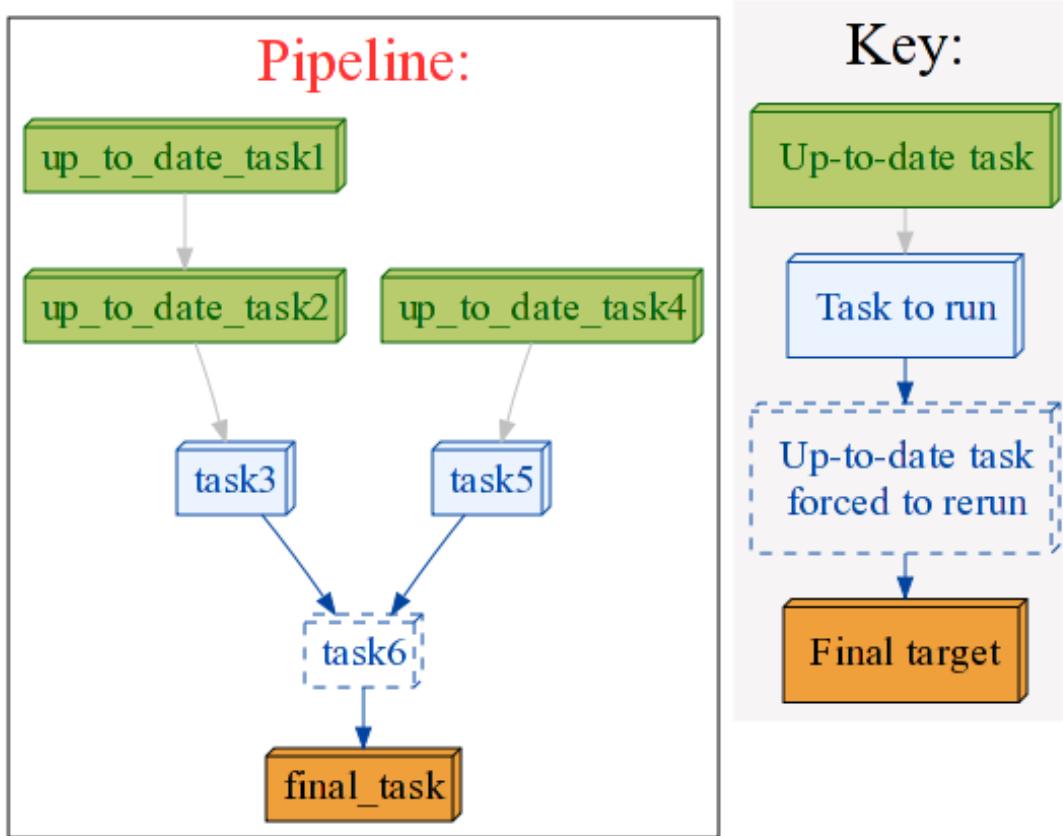
## 5.14 Chapter 12: *Checking dependencies to run tasks in order*

- *Manual overview*

How does **Ruffus** decide how to run your pipeline?

- In which order should pipelined functions be called?
- Which parts of the pipeline are up-to-date and do not need to be rerun?

### 5.14.1 Running all out-of-date tasks and dependents



By default, *ruffus* will

- build a flow chart (dependency tree) of pipelined tasks (functions)
- start from the most ancestral tasks with the fewest dependencies (`task1` and `task4` in the flowchart above).
- walk up the tree to find the first incomplete / out-of-date tasks (i.e. `task3` and `task5`).
- start running from there

**All down-stream (dependent) tasks will be re-run anyway, so we don't have to test** whether they are up-to-date or not.

---

**Note:** This means that **ruffus** *may* ask any task if their jobs are out of date more than once:

- once when deciding which parts of the pipeline have to be run
- once just before executing the task.

---

*Ruffus* tries to be clever / efficient, and does the minimal amount of querying.

### 5.14.2 A simple example

Four successive tasks to run:

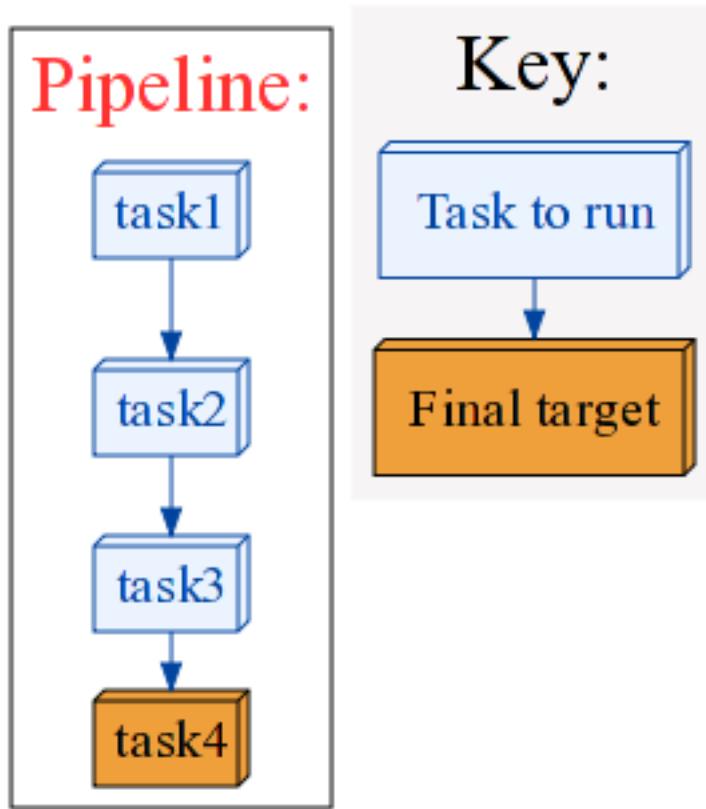
---

Note: The full code is available [here](#).

---

Suppose we have four successive tasks to run, whose flowchart we can print out by running:

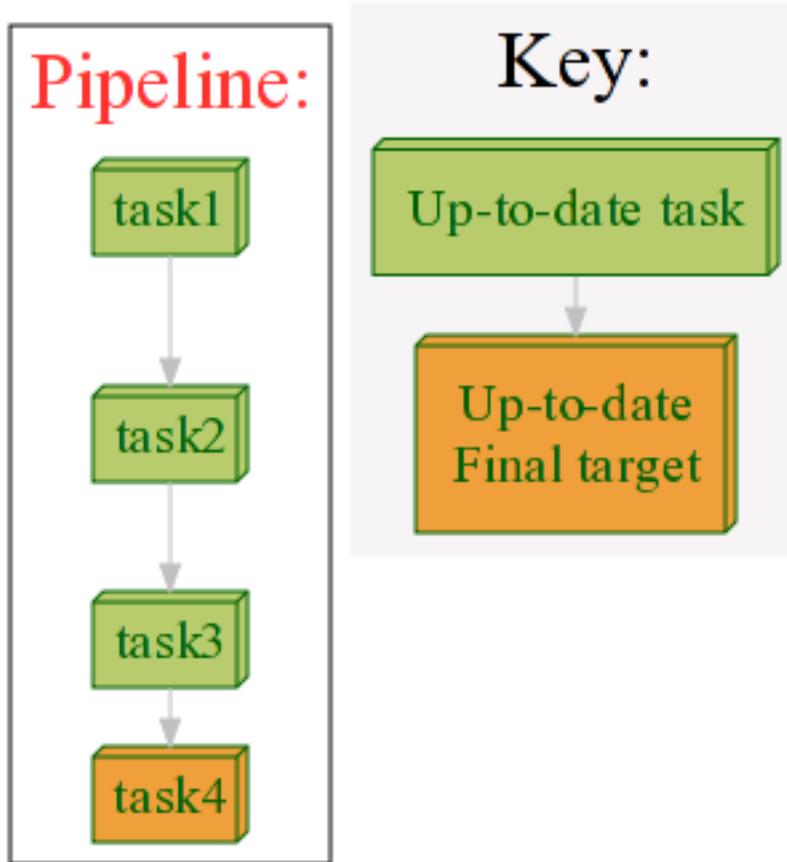
```
pipeline_printout_graph ("flowchart.png", "png", [task4],  
                        draw_vertically = True)
```



We can see that all four tasks need to run reach the target task4.

**Pipeline tasks are up-to-date:**

After the pipeline runs (`python simpler.py -d ""`), all tasks are up to date and the flowchart shows:



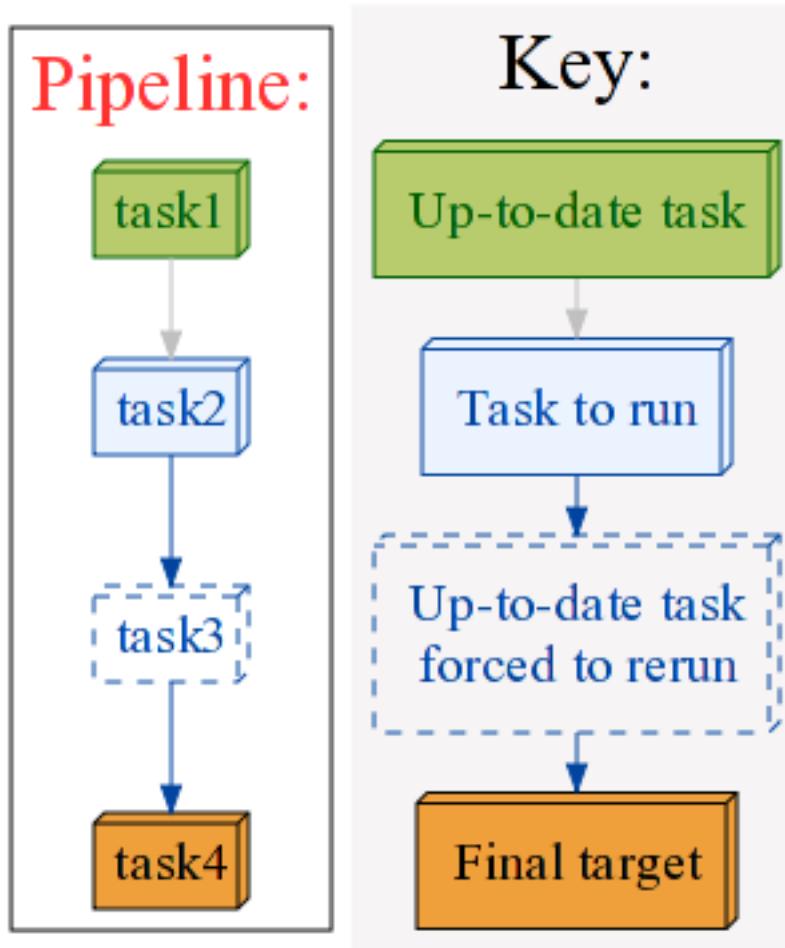
### Some tasks out of date:

If we then made task2 and task4 out of date by modifying their *inputs* files:

```
open("a.1", "w")
open("a.3", "w")
```

the flowchart would show:

1. the pipeline only has to rerun from task2.
2. task1 is complete / up-to-date
3. task3 will have to be re-run because it follows (depends on) task2.



### 5.14.3 Forced Reruns

Even if a pipeline stage appears to be up to date, you can always force the pipeline to include from one or more task functions.

This is particularly useful, for example, if the pipeline data hasn't changed but the analysis or computational code has.

```
pipeline_run([task4], [task1])
```

will run all tasks from task1 to task4

Both the “target” and the “forced” lists can include as many tasks as you wish. All dependencies are still carried out and out-of-date jobs rerun.

### 5.14.4 Esoteric option: Minimal Reruns

In the above example, you could point out that task3 is not out of date. And if we were only interested in the immediate dependencies or prerequisites leading up to task4, we might not need task2 to rerun at all, only task4.

This rather dangerous option is useful if you don't want to keep all the intermediate files/results from upstream tasks. The pipeline will only not involve any incomplete tasks which precede an up-to-date result.

This is seldom what you intend, and you should always check that the appropriate stages of the pipeline are executed in the flowchart output.

In such cases, we can rerun the pipeline with the following option:

```
pipeline_run([task4], gnu_make_maximal_rebuild_mode = False)
```

and only task4 will rerun.

## 5.15 Chapter 13: Generating parameters on the fly with @files

- *Manual overview*
- *@files on-the-fly syntax in detail*

Sometimes, it is necessary, or perhaps more convenient, to generate parameters on the fly or at runtime. This powerful ability to generate the exact parameters you need is sometimes worth the slight increase in complexity.

### 5.15.1 @files

To generate parameters on the fly, pass the decorator **files** with a *generator* function which yields one list / tuple of parameters per job. For example:

```
from ruffus import *
def generate_parameters_on_the_fly():
    """
    returns one list of parameters per job
    """
    parameters = [
        ['A.input', 'A.output', (1, 2)], # 1st job
        ['B.input', 'B.output', (3, 4)], # 2nd job
        ['C.input', 'C.output', (5, 6)], # 3rd job
    ]
    for job_parameters in parameters:
        yield job_parameters

@files(generate_parameters_on_the_fly)
def pipeline_task(input, output, extra):
    open(output, "w").write(open(input).read())
    sys.stderr.write("%d + %d => %d\n" % (extra[0], extra[1], extra[0] + extra[1]))

pipeline_run([pipeline_task])
```

Produces:

```
Task = parallel_task
1 + 2 = 3
Job = ["A", 1, 2] completed
3 + 4 = 7
```

```
Job = ["B", 3, 4] completed
5 + 6 = 11
Job = ["C", 5, 6] completed
```

---

**Note:** Be aware that the parameter generating function may be invoked *more than once*:

The first time to check if this part of the pipeline is up-to-date.

The second time when the pipeline task function is run.

---

The resulting *inputs*, *outputs* and any additional extra parameters per job are treated normally for the purposes of checking to see if jobs are up-to-date and need to be re-run.

## 5.15.2 Permutations and Combinations

The *accompanying example* provides a more realistic reason why you would want to generate parameters on the fly. It is a fun piece of code, which generates N x M combinations from two sets of files as the *inputs* of a pipeline stage.

The *inputs / outputs* filenames are generated as a pair of nested for-loops to produce the N (outside loop) x M (inside loop) combinations, with the appropriate parameters for each job yielded per iteration of the inner loop. The gist of this is:

```
# _____
# Step 1:
#
#      N x M jobs
#
def generate_simulation_params () :
    """
    Custom function to generate
    file names for gene/gwas simulation study
    """
    for sim_file in get_simulation_files():
        for (gene, gwas) in get_gene_gwas_file_pairs():
            result_file = "%s.%s.results" % (gene, sim_file)
            yield (gene, gwas, sim_file), result_file
```

```
@files(generate_simulation_params)
def gwas_simulation(input_files, output_file):
    ...
```

If `get_gene_gwas_file_pairs()` produces:

```
['a.sim', 'b.sim', 'c.sim']
```

and `get_gene_gwas_file_pairs()` produces:

```
[('1.gene', '1.gwas'), ('2.gene', '2.gwas')]
```

then we would end up with 3 x 2 = 6 jobs and the following equivalent function calls:

```
gwas_simulation(('1.gene', '1.gwas', 'a.sim'), "1.gene.a.sim.results")
gwas_simulation(('2.gene', '2.gwas', 'a.sim'), "2.gene.a.sim.results")
gwas_simulation(('1.gene', '1.gwas', 'b.sim'), "1.gene.b.sim.results")
```

```
gwas_simulation('2.gene', '2.gwas', 'b.sim'), "2.gene.b.sim.results")
gwas_simulation('1.gene', '1.gwas', 'c.sim'), "1.gene.c.sim.results")
gwas_simulation('2.gene', '2.gwas', 'c.sim'), "2.gene.c.sim.results")
```

The *accompanying code* looks slightly more complicated because of some extra bookkeeping.

## 5.16 Chapter 14: @collate: group together disparate input into sets of results

- *Manual overview*
- *@collate syntax in detail*

It is often very useful to group together disparate *inputs* into several categories, each of which lead to a separate *output*. In the example shown below, we produce separate summaries of results depending on which species the file belongs to.

**Ruffus** uses the term `collate` in a rough analogy to the way printers group together copies of documents appropriately.

### 5.16.1 Collating many *inputs* each into a single *output*

Our example starts with some files which presumably have been created by some earlier stages of our pipeline. We simulate this here with this code:

```
files_names = [
    "mammals.tiger.wild.animals"
    "mammals.lion.wild.animals"
    "mammals.lion.handreared.animals"
    "mammals.dog.tame.animals"
    "mammals.dog.wild.animals"
    "mammals.dog.feral.animals"
    "reptiles.crocodile.wild.animals" ]
for f in files_names:
    open(f, "w").write(f)
```

However, we are only interested in mammals, and we would like the files of each species to end up in its own directory, i.e. tiger, lion and dog:

```
import os
os.mkdir("tiger")
os.mkdir("lion")
os.mkdir("dog")
```

Now we would like to place each file in a different destination, depending on its species. The following regular expression marks out the species name `r'mammals.([^.]+)'`. For `mammals.tiger.wild.animals`, the first matching group `(\1) == "tiger"`

Then, the following:

```
from ruffus import *

@collate('*.animals',
         regex(r'mammals.([^.]+)'),      # inputs = all *.animal files
         r'\1/animals.in_my_zoo',        # regular expression
         r'\1')                         # single output file per species
                                 # species name

def capture_mammals(infiles, outfile, species):
```

```
# summarise all animals of this species
print "Collating %s" % species

o = open(outfile, "w")
for i in infilenames:
    o.write(open(infile).read() + "\ncaptured\n")

pipeline_run([capture_mammals])
```

puts each captured mammal in its own directory:

```
Task = capture_mammals
Job = [ (mammals.lion.handreared.animals, mammals.lion.wild.animals) -> lion/animals.in_my_zoo
Job = [ (mammals.tiger.wild.animals, ) -> tiger/animals.in_my_zoo] completed
Job = [ (mammals.dog.tame.animals, mammals.dog.wild.animals, mammals.dog.feral.animals) -> dog/animals.in_my_zoo]
```

The crocodile has been discarded because it isn't a mammal and the file name doesn't match the mammal part of the regular expression.

## 5.17 Chapter 15: add\_inputs() and inputs(): *Controlling both input and output files with @transform*

- *Manual overview*
- *@transform* syntax in detail

The standard *@transform* allows you to send a list of data files to the same pipelined function and for the resulting *outputs* parameter to be automatically inferred from file names in the *inputs*.

**There are two situations where you might desire additional flexibility:**

1. You need to add additional prerequisites or filenames to the *inputs* of every single one of your jobs
2. (Less often,) the actual *inputs* file names are some variant of the *outputs* of another task.

Either way, it is occasionally very useful to be able to generate the actual *inputs* as well as *outputs* parameters by regular expression substitution. The following examples will show you both how and why you would want to do this.

### 5.17.1 Adding additional *input* prerequisites per job

#### 1.) Example: compiling c++ code

Suppose we wished to compile some c++ ("\*.cpp") files:

```
source_files = "hasty.cpp", "tasty.cpp", "messy.cpp"
for source_file in source_files:
    open(source_file, "w")
```

The ruffus code would look like this:

```
from ruffus import *

@transform(source_files, suffix(".cpp"), ".o")
def compile(input_filename, output_file_name):
    open(output_file_name, "w")
```

This results in the following jobs:

```
>>> pipeline_run([compile], verbose = 2, multiprocess = 3)

Job = [None -> hasty.cpp] completed
Job = [None -> tasty.cpp] completed
Job = [None -> messy.cpp] completed
Completed Task = prepare_cpp_source

Job = [hasty.cpp -> hasty.o] completed
Job = [messy.cpp -> messy.o] completed
Job = [tasty.cpp -> tasty.o] completed
Completed Task = compile
```

## 2.) Example: Adding a header file with add\_inputs(..)

All this is plain vanilla `@transform` syntax. But suppose that we need to add a common header file "universal.h" to our compilation. The `add_inputs` provides for this with the minimum of fuss:

```
# create header file
open("universal.h", "w")

# compile C++ files with extra header
@transform(prepare_cpp_source, suffix(".cpp"), add_inputs("universal.h"), ".o")
def compile(input_filename, output_file_name):
    open(output_file_name, "w")
```

Now the input file is a python list, with "universal.h" added to each "\*.cpp"

```
>>> pipeline_run([compile], verbose = 2, multiprocess = 3)

Job = [ [hasty.cpp, universal.h] -> hasty.o] completed
Job = [ [messy.cpp, universal.h] -> messy.o] completed
Job = [ [tasty.cpp, universal.h] -> tasty.o] completed
Completed Task = compile
```

## 5.17.2 Additional *input* prerequisites can be globs, tasks or pattern matches

A common requirement is to include the corresponding header file in compilations. It is easy to use `add_inputs` to look up additional files via pattern matches.

## 3.) Example: Adding matching header file

To make this example more fun, we shall also:

1. Give each source code file its own ordinal
2. Use `add_inputs` to add files produced by another task function

```
# each source file has its own index
source_names = [("hasty.cpp", 1),
                ("tasty.cpp", 2),
                ("messy.cpp", 3), ]
header_names = [sn.replace(".cpp", ".h") for (sn, i) in source_names]
header_names.append("universal.h")
```

```
#  
#   create header and source files  
#  
for source, source_index in source_names:  
    open(source, "w")  
  
for header in header_names:  
    open(header, "w")  
  
  
from ruffus import *  
  
#  
#   lookup embedded strings in each source files  
#  
@transform(source_names, suffix(".cpp"), ".embedded")  
def get_embedded_strings(input_filename, output_file_name):  
    open(output_file_name, "w")  
  
  
# compile C++ files with extra header  
@transform(source_names, suffix(".cpp"),  
           add_inputs("universal.h",  
                      r"\1.h",  
                      get_embedded_strings), ".o")  
def compile(input_params, output_file_name):  
    open(output_file_name, "w")  
  
  
pipeline_run([compile], verbose = 2, multiprocess = 3)
```

This script gives the following output

```
>>> pipeline_run([compile], verbose = 2, multiprocess = 3)  
  
Job = [[hasty.cpp, 1] -> hasty.embedded] completed  
Job = [[messy.cpp, 3] -> messy.embedded] completed  
Job = [[tasty.cpp, 2] -> tasty.embedded] completed  
Completed Task = get_embedded_strings  
  
Job = [[[hasty.cpp, 1],  
        universal.h,  
        hasty.h,  
        hasty.embedded, messy.embedded, tasty.embedded]  
      -> hasty.o] completed  
# inputs  
# common header  
# corresponding header  
# output of get_embedded_strings  
Job = [[[messy.cpp, 3],  
        universal.h,  
        messy.h,  
        hasty.embedded, messy.embedded, tasty.embedded]  
      -> messy.o] completed  
# inputs  
# common header  
# corresponding header  
# output of get_embedded_strings  
Job = [[[tasty.cpp, 2],  
        universal.h,  
        tasty.h,  
        hasty.embedded, messy.embedded, tasty.embedded]  
      -> tasty.o] completed  
# inputs  
# common header  
# corresponding header  
# output of get_embedded_strings  
Completed Task = compile
```

---

We can see that the `compile(...)` task now has four sets of *inputs*:

1. The original inputs (e.g. `[hasty.cpp, 1]`)

And three additional added by `add_inputs(...)`

2. A header file (`universal.h`) common to all jobs
3. The matching header (e.g. `hasty.h`)
4. The output from another task `get_embedded_strings()` (e.g. `hasty.embedded`, `messy.embedded`, `tasty.embedded`)

---

**Note:** For input parameters with nested structures (lists or sets), the pattern matching is on the first filename string Ruffus comes across (DFS).

So for `["hasty.c", 0]`, the pattern matches `"hasty.c"`.

If in doubt, use `pipeline_printout` to check what parameters Ruffus is using.

---

#### 4.) Example: Using `regex(..)` instead of `suffix(..)`

Suffix pattern matching is much simpler and hence is usually preferable to the more powerful regular expressions. We can rewrite the above example to use `regex` as well to give exactly the same output.

```
# compile C++ files with extra header
@transform(source_names, regex(r"(.+)\.cpp"),
           add_inputs( "universal.h",
                       r"\1.h",
                       get_embedded_strings      ), r"\1.o")
def compile(input_params, output_file_name):
    open(output_file_name, "w")
```

---

**Note:** The backreference `\g<0>` usefully substitutes the entire substring matched by the regular expression.

---

### 5.17.3 Replacing all input parameters with `inputs(...)`

More rarely, it is necessary to replace all the input parameters wholesale.

#### 4.) Example: Running matching python scripts

In the following example, we are not compiling C++ source files but invoking corresponding python scripts which have the same name.

Given three c++ files and their corresponding python scripts:

```
# each source file has its own index
source_names = [ ("hasty.cpp", 1),
                 ("tasty.cpp", 2),
                 ("messy.cpp", 3), ]

#
#   create c++ source files and corresponding python files
#
for source, source_index in source_names:
```

```
open(source, "w")
open(source.replace(".cpp", ".py"), "w")
```

The Ruffus code will call each python script corresponding to their c++ counterpart:

```
from ruffus import *

# run corresponding python files
@transform(source_names, suffix(".cpp"), inputs(r"\1.py"), ".results")
def run_python_file(input_params, output_file_name):
    open(output_file_name, "w")

pipeline_run([run_python_file], verbose = 2, multiprocess = 3)
```

**Resulting in this output:**

```
>>> pipeline_run([run_python_file], verbose = 2, multiprocess = 3)
Job = [hasty.py -> hasty.results] completed
Job = [messy.py -> messy.results] completed
Job = [tasty.py -> tasty.results] completed
Completed Task = run_python_file
```

## 5.) Example: Using regex instead of suffix

Again, the same code can be written (less clearly) using the more powerful **regex** and python regular expressions:

```
from ruffus import *

# run corresponding python files
@transform(source_names, regex(r"(+)\.cpp"), inputs(r"\1.py"), r"\1.results")
def run_python_file(input_params, output_file_name):
    open(output_file_name, "w")

pipeline_run([run_python_file], verbose = 2, multiprocess = 3)
```

This is about as sophisticated as **@transform** ever gets!

## 5.18 Chapter 16: *Esoteric: Running jobs in parallel without using files with @parallel*

- *Manual overview*
- **@parallel** syntax in detail

### 5.18.1 **@parallel**

**@parallel** supplies parameters for multiple **jobs** exactly like **@files** except that:

1. The first two parameters are not treated like *inputs* and *outputs* parameters, and strings are not assumed to be file names

2. Thus no checking of whether each job is up-to-date is made using *inputs* and *outputs* files
3. No expansions of *glob* patterns or *output* from previous tasks is carried out.

This syntax is most useful when a pipeline stage does not involve creating or consuming any files, and you wish to forego the conveniences of *@files*, *@transform* etc.

The following code performs some arithmetic in parallel:

```
import sys
from ruffus import *
parameters = [
    ['A', 1, 2], # 1st job
    ['B', 3, 4], # 2nd job
    ['C', 5, 6], # 3rd job
]
@parallel(parameters)
def parallel_task(name, param1, param2):
    sys.stderr.write("    Parallel task %s: " % name)
    sys.stderr.write("%d + %d = %d\n" % (param1, param2, param1 + param2))

pipeline_run([parallel_task])
```

produces the following:

```
Task = parallel_task
Parallel task A: 1 + 2 = 3
Job = ["A", 1, 2] completed
Parallel task B: 3 + 4 = 7
Job = ["B", 3, 4] completed
Parallel task C: 5 + 6 = 11
Job = ["C", 5, 6] completed
```

## 5.19 Chapter 17: *Writing custom functions to decide which jobs are up to date*

- *Manual overview*
- *@check\_if\_uptodate syntax in detail*

### 5.19.1 @check\_if\_uptodate : Manual dependency checking

tasks specified with

- *@files*
- *@split*
- *@transform*
- *@merge*
- *@collate*

have automatic dependency checking based on file modification times.

Sometimes, you might want to decide have more control over whether to run jobs, especially if a task does not rely on or produce files (i.e. with *@parallel*)

You can write your own custom function to decide whether to run a job. This takes as many parameters as your task function, and needs to return a tuple for whether an update is required, and why (i.e. tuple(bool, str))

This simple example which creates the file "a.1" if it does not exist:

```
from ruffus import *
@files(None, "a.1")
def create_if_necessary(input_file, output_file):
    open(output_file, "w")

pipeline_run([create_if_necessary])
```

could be rewritten more laboriously as:

```
from ruffus import *
import os
def check_file_exists(input_file, output_file):
    if os.path.exists(output_file):
        return False, "File already exists"
    return True, "%s is missing" % output_file

@parallel([[None, "a.1"]])
@check_if_upToDate(check_file_exists)
def create_if_necessary(input_file, output_file):
    open(output_file, "w")

pipeline_run([create_if_necessary])
```

**Both produce the same output:**

```
Task = create_if_necessary
Job = [null, "a.1"] completed
```

---

**Note:** The function specified by `@check_if_update` can be called more than once for each job.

See the *description here* of how **Ruffus** decides which tasks to run.

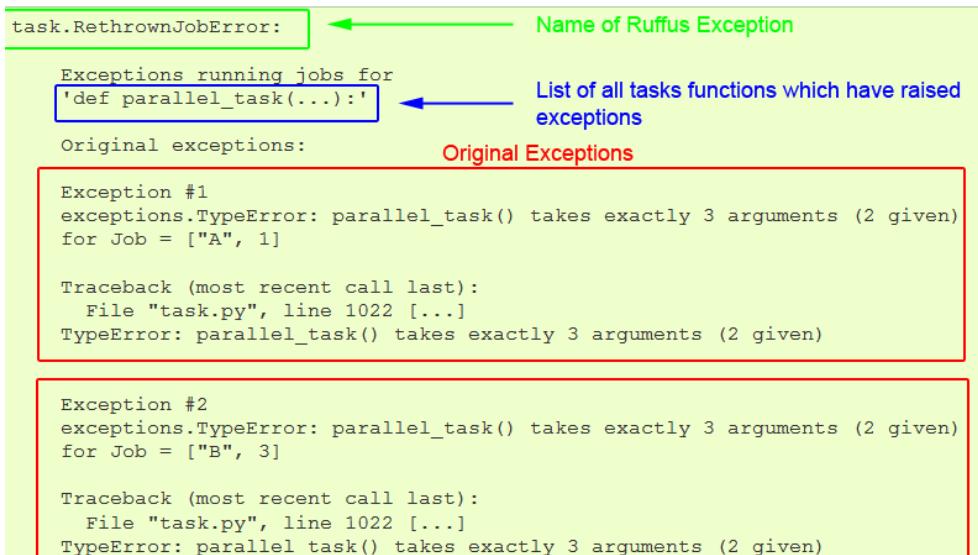
---

## 5.20 Chapter 18: *Exceptions thrown inside a pipeline*

- *Manual overview*

The goal for **Ruffus** is that exceptions should just work *out-of-the-box* without any fuss. This is especially important for exceptions that come from your code which may be raised in a different process. Often multiple parallel operations (jobs or tasks) fail at the same time. **Ruffus** will forward each of these exceptions with the tracebacks so you can jump straight to the offending line.

This example shows separate exceptions from two jobs running in parallel:



### 5.20.1 Multiple Errors

For any task where exceptions are thrown, *Ruffus* will continue executing all the jobs currently in progress (up to the maximum number of concurrent jobs (`multiprocess`) set in `pipeline_run`). Each of these may raise separate exceptions. This seems a fair tradeoff between being able to gather detailed error information for running jobs, and not wasting too much time for a task that is going to fail anyway.

### 5.20.2 Interrupting the pipeline

If your task function raises a `Ruffus.JobSignalledBreak` Exception, this will immediately halt the pipeline at that point, without waiting for other jobs in the queue to complete:

```
from ruffus import *
@parallel([['A', 1], ['B', 3]])
def parallel_task(name, param1):
    if name == 'A': return False

pipeline_run([parallel_task])
```

produces the following (abbreviated):

```
task.RethrownJobError:

Exceptions running jobs for
'def parallel_task(...):'

Original exception:

Exception #1
task.JobSignalledBreak: Job = ["A", 1] returned False
for Job = ["A", 1]
```

## 5.21 Chapter 19: Logging progress through a pipeline

- *Manual overview*

There are two parts to logging with **Ruffus**:

- Logging progress through the pipeline

This produces the sort of output displayed in this manual:

```
>>> pipeline_run([parallel_io_task])
Task = parallel_io_task
    Job = ["a.1" -> "a.2", "A file"] completed
    Job = ["b.1" -> "b.2", "B file"] unnecessary: already up to date
Completed Task = parallel_io_task
```

- Logging your own messages from within your pipelined functions.

Because **Ruffus** may run these in separate process (multiprocessing), some attention has to be paid to how to send and synchronise your log messages across process boundaries.

We shall deal with these in turn.

### 5.21.1 Logging task/job completion

By default, *Ruffus* logs each task and each job as it is completed to `sys.stderr`.

`pipeline_run()` includes an optional `logger` parameter which defaults to `stderr_logger`. Set this to `black_hole_logger` to turn off all tracking messages as the pipeline runs:

```
pipeline_run([pipelined_task], logger = black_hole_logger)
```

### Controlling logging verbosity

`pipeline_run()` currently has five levels of verbosity, set by the optional `verbose` parameter which defaults to 1:

```
verbose = 0: nothing
verbose = 1: logs completed jobs/tasks;
verbose = 2: logs up to date jobs in incomplete tasks
verbose = 3: logs reason for running job
verbose = 4: logs messages useful only for debugging ruffus pipeline code
```

Verbose > 2 are intended for debugging **Ruffus** by the developers and the details are liable to change from release to release

### Using your own logging

You can specify your own logging by providing a log object to `pipeline_run()`. This log object should have `debug()` and `info()` methods.

Instead of writing your own, it is usually more convenient to use the python `logging` module which provides logging classes with rich functionality. The following sets up a logger to a rotating set of files:

```
import logging
import logging.handlers

LOG_FILENAME = '/tmp/ruffus.log'

# Set up a specific logger with our desired output level
my_ruffus_logger = logging.getLogger('My_Ruffus_logger')
my_ruffus_logger.setLevel(logging.DEBUG)

# Add the log message handler to the logger
handler = logging.handlers.RotatingFileHandler(
    LOG_FILENAME, maxBytes=2000, backupCount=5)

my_ruffus_logger.addHandler(handler)

from ruffus import *

@files(None, "a.1")
def create_if_necessary(input_file, output_file):
    """Description: Create the file if it does not exists"""
    open(output_file, "w")

pipeline_run([create_if_necessary], [create_if_necessary], logger=my_ruffus_logger)
print open("/tmp/ruffus.log").read()
```

The contents of `/tmp/ruffus.log` are as specified:

```
Task = create_if_necessary
Description: Create the file if it does not exists
Job = [null -> "a.1"] completed
```

## 5.21.2 Your own logging *within* each job

It is often useful to log the messages from within each of your pipelined functions.

However, each job runs in a separate process, and it is *not* a good idea to pass the logging object itself between jobs:

1. logging is not synchronised between processes
2. logging objects can not be pickled and sent across processes

The best thing to do is to have a centralised log and to have each job invoke the logging methods (e.g. `debug`, `warning`, `info` etc.) across the process boundaries in the centralised log.

The `Ruffus proxy_logger` module provides an easy way to share `logging` objects among jobs. This requires just two simple steps:

---

**Note:** The full code shows how this can be coded.

---

### 1. Set up log from config file

```
from ruffus.proxy_logger import *
(logger_proxy,
logging_mutex) = make_shared_logger_and_proxy (setup_std_shared_logger,
```

```
"my_logger",
{"file_name" :"/my/lg.log"})
```

## 2. Give each job proxy to logger

Now, pass:

- `logger_proxy` (which forwards logging calls across jobs) and
- `logging_mutex` (which prevents different jobs which are logging simultaneously from being jumbled up)

to each job:

```
@files(None, 'a.1', logger_proxy, logging_mutex)
def task1(ignore_infile, outfile, logger_proxy, logging_mutex):
    """
    Log within task
    """
    open(outfile, "w").write("Here we go")
    with logging_mutex:
        logger_proxy.info("Here we go logging")
```

## 5.22 Chapter 20: `@files_re`: Deprecated syntax using regular expressions

- *Manual overview*
- `@files_re` syntax in detail

### 5.22.1 `@files_re`

This is older, now deprecated syntax.

`@files_re` combines the functionality of `@transform`, `@collate` and `@merge` in one overloaded decorator.

This is the reason why its use is discouraged. `@files_re` syntax is far too overloaded and context-dependent to support its myriad of different functions.

The following documentation is provided to help maintain historical **ruffus** usage.

#### Transforming input and output filenames

For example, the following code from **Chapter 8** takes files from the previous pipeline task, and makes new output parameters with the `.sums` suffix in place of the `.chunks` suffix:

```
@transform(step_4_split_numbers_into_chunks, suffix(".chunks"), ".sums")
def step_5_calculate_sum_of_squares (input_file_name, output_file_name):
    """
    calculate sums and sums of squares for all values in the input_file_name
    writing to output_file_name
    """
```

This can be written using `@files_re` equivalently:

```
@files_re(step_4_split_numbers_into_chunks, r".chunks", r".sums")
def step_5_calculate_sum_of_squares (input_file_name, output_file_name):
    """
```

## Collating many *inputs* into a single *output*

Similarly, the following code from `lmanual.collate.chapter_numl` collects **inputs** from the same species in the same directory:

```
@collate('*.animals',
          regex(r'mammals.([^.]+)'),           # inputs = all *.animal files
          r'\1/animals.in_my_zoo',             # regular expression
          r'\1')                             # single output file per species
                                         # species name
def capture_mammals(infiles, outfile, species):
    # summarise all animals of this species
    """
```

This can be written using `@files_re` equivalently using the `combine` indicator:

```
@files_re('*.animals',
          regex(r'mammals.([^.]+)'),           # inputs = all *.animal files
          combine(r'\1/animals.in_my_zoo'),     # regular expression
          r'\1')                             # single output file per species
                                         # species name
def capture_mammals(infiles, outfile, species):
    # summarise all animals of this species
    """
```

## Generating *input* and *output* parameter using regular expressions

The following code generates additional *Input* prerequisite file names which match the original *Input* files names.

We want each job of our `analyse()` function to get corresponding pairs of `xx.chunks` and `xx.red_indian` files when

```
*.chunks are generated by the task function split_up_problem() and
*.red_indian are generated by the task function make_red_ians():

@follows(make_red_ians)
@transform(split_up_problem,
          regex(r"(.*)chunks"),
          inputs([r"\g<0>",
                  r"\1.red_indian"]),
          r"\1.results"
          )
def analyse(input_filenames, output_file_name):
    "Do analysis here"
```

The equivalent code using `@files_re` looks very similar:

```
@follows(make_red_ians)
@files_re( split_up_problem,
          r"(.*)chunks",
          [r"\g<0>",
           r"\1.red_indian"],
          r"\1.results")
```

```
def analyse(input_filenames, output_file_name):
    "Do analysis here"
```

## 5.23 Recipes

### 5.23.1 General

## 5.24 A simple tutorial: 8 steps to *Ruffus*

### 5.24.1 Table of Contents

#### Features

The **Ruffus** provides automatic support for

- Managing dependencies
- Parallel jobs
- Re-starting from arbitrary points, especially after errors
- Display of the pipeline as a flowchart
- Reporting

This tutorial has seven steps which cover all the core functionality of *Ruffus*.

Don't worry if steps 1 and 2 seem a bit slow: Once you get used to **Ruffus** steps 4-8 will be a breeze.

You can click on “previous” and “next” at the top and bottom of each page to navigate through the tutorial.

#### The first steps (1-4)

The first half of the tutorial will show you how to:

1. *Chain tasks (functions) together into a pipeline*
2. *Provide parameters to run jobs in parallel*
3. *Tracing through your new pipeline*
4. *Using flowcharts*

#### A worked example (steps 5-8)

The second half of the tutorial is a worked example to calculate the sample variance of 10,000 random numbers. This shows you how to:

5. *Split up a large problem into smaller chunks*
6. *Calculate partial solutions in parallel*
7. *Re-combine the partial solutions into the final result*

## 8. Automatically signal the completion of each step of our pipeline

This covers the core functionality of **Ruffus**.

- Simple tutorial overview

# 5.25 Step 1: An introduction to Ruffus pipelines

## 5.25.1 Overview

Computational pipelines transform your data in stages until the final result is produced. One easy way to understand pipelines is by imagining your data flowing across a series of pipes until it reaches its final destination. Even quite complicated processes can be simplified if we broke things down into simple stages. Of course, it helps if we can visualise the whole process.

Ruffus is a way of automating the plumbing in your pipeline: You supply the python functions which perform the data transformation, and tell Ruffus how these pipeline `task` functions are connected up. Ruffus will make sure that the right data flows down your pipeline in the right way at the right time.

---

**Note:** Ruffus refers to each stage of your pipeline as a *task*.

---

## 5.25.2 A gentle introduction to Ruffus syntax

Let us start with the usual “Hello World” programme.

We have the following two python functions which we would like to turn into an automatic pipeline:

```
def first_task():
    print "Hello "

def second_task():
    print "world"
```

The simplest **Ruffus** pipeline would look like this:

The functions which do the actual work of each stage of the pipeline remain unchanged. The role of **Ruffus** is to make sure these functions are called in the right order, with the right parameters, running in parallel using multiprocessing if desired.

There are three simple parts to building a **ruffus** pipeline

1. importing ruffus
2. “Decorating” functions which are part of the pipeline
3. Running the pipeline!

## 5.25.3 “Decorators”

You need to tag or *decorator* existing code to tell **Ruffus** that they are part of the pipeline.

---

**Note:** python *decorators* are ways to tag or mark out functions.

They start with a @ prefix and take a number of parameters in parenthesis.

---

The **ruffus** decorator `@follows` makes sure that `second_task` follows `first_task`.

Multiple *decorators* can be used for each *task* function to add functionality to *Ruffus* pipeline functions. However, the decorated python functions can still be called normally, outside of *Ruffus*. *Ruffus decorators* can be added to (stacked on top of) any function in any order.

- *More on @follows in the in the Ruffus ‘Manual’*
- *@follows syntax in detail*

#### 5.25.4 Running the pipeline

We run the pipeline by specifying the **last** stage (*task* function) of your pipeline. Ruffus will know what other functions this depends on, following the appropriate chain of dependencies automatically, making sure that the entire pipeline is up-to-date.

Because `second_task` depends on `first_task`, both functions are executed in order.

```
>>> pipeline_run([second_task], verbose = 1)
```

Ruffus by default prints out the `verbose` progress through the pipelined code, interleaved with the **Hello** printed by `first_task` and **World** printed by `second_task`.

### 5.26 Step 2: @transform-ing data in a pipeline

- *Simple tutorial overview*
- *@transform syntax in detail*

---

**Note:** Remember to look at the example code:

- *Python Code for step 2*
- 

#### 5.26.1 Overview

Computational pipelines transform your data in stages until the final result is produced. Ruffus automates the plumbing in your pipeline. You supply the python functions which perform the data transformation, and tell Ruffus how these pipeline stages or *task* functions are connected together.

---

**Note:** The best way to design a pipeline is to:

- **write down the file names of the data as it flows across your pipeline**
  - **write down the names of functions which transforms the data at each stage of the pipeline.**
- 

By letting **Ruffus** manage your pipeline parameters, you will get the following features for free:

1. only out-of-date parts of the pipeline will be re-run
2. multiple jobs can be run in parallel (on different processors if possible)
3. pipeline stages can be chained together automatically

## 5.26.2 @transform

Let us start with the simplest pipeline with a single *input* data file **transformed** into a single *output* file. We will add some arbitrary extra parameters as well.

The `@transform` decorator tells Ruffus that this task function **transforms** each and every piece of input data into a corresponding output.

In other words, inputs and outputs have a **1 to 1** relationship.

---

**Note:** In the second part of the tutorial, we will encounter more decorators which can *split up*, or *join together* or *group* inputs.

In other words, inputs and output can have **many to one**, **many to many** etc. relationships.

---

Let us provide **inputs** and **outputs** to our new pipeline:

The `@transform` decorator tells Ruffus to generate the appropriate arguments for our python function:

- The input file name is as given: `job1.input`
- The output file name is the input file name with its **suffix** of `.input` replaced with `.output1`
- There are two extra parameters, a string and a number.

This is exactly equivalent to the following function call:

```
first_task('job1.input', 'job1.output1', "some_extra.string.for_example", 14)
```

Even though this (empty) function doesn't do anything just yet, the output from **Ruffus pipeline\_run** will show that that this part of the pipeline completed successfully:

---

## 5.26.3 Task functions as recipes

This may seem like a lot of effort and complication for something so simple: a normal python function call. However, now that we have annotated a task, we can start using it as part of our computational pipeline:

Each *task* function of the pipeline is a recipe or `rule` which can be applied repeatedly to our data.

For example, one can have

- a `compile()` *task* which will compile any number of source code files, or
- a `count_lines()` *task* which will count the number of lines in any file or
- an `align_dna()` *task* which will align the DNA of many chromosomes.

---

**Note: Key Ruffus Terminology:**

A *task* is an annotated python function which represents a recipe or stage of your pipeline.

A *job* is each time your recipe is applied to a piece of data, i.e. each time Ruffus calls your function.

Each **task** or pipeline recipe can thus have many **jobs** each of which can work in parallel on different data.

---

In the original example, we have made a single output file by supplying a single input parameter. We shall use much the same syntax to apply the same recipe to *multiple* input files. Instead of providing a single *input*, and a single *output*, we are going to specify the parameters for *three* jobs at once:

```
# previously,
# first_task_params = 'job1.input'
first_task_params = [
    'job1.input',
    'job2.input',
    'job3.input'
]

# make sure the input files are there
open('job1.input', "w")
open('job2.input', "w")
open('job3.input', "w")

pipeline_run([first_task])
```

Just by changing the inputs from a single file to a list of three files, we now have a pipeline which runs independently on three pieces of data. The results should look familiar:

```
>>> pipeline_run([first_task])
Job   = [job1.input -> job1.output1,
         some_extra.string.for_example, 14] completed
Job   = [job2.input -> job2.output1,
         some_extra.string.for_example, 14] completed
Job   = [job3.input -> job3.output1,
         some_extra.string.for_example, 14] completed
Completed Task = first_task
```

## 5.26.4 Multiple steps

Best of all, it is easy to add another step to our initial pipeline.

We have to

- add another `@transform` decorated function (`second_task()`),
- specify `first_task()` as the source:
- use a suffix which matches the output from `first_task()`

```
@transform(first_task, suffix(".output1"), ".output2")
def second_task(input_file, output_file):
    # make output file
    open(output_file, "w")
```

- call `pipeline_run()` with the correct final task (`second_task()`)

The full source code can be found [here](#)

With very little effort, we now have three independent pieces of information coursing through our pipeline. Because `second_task()` *transforms* the output from `first_task()`, it magically knows its dependencies and that it too has to work on three jobs.

## 5.26.5 Multi-tasking

Though, three jobs have been specified in parallel, **Ruffus** defaults to running each of them successively. With modern CPUs, it is often a lot faster to run parts of your pipeline in parallel, all at the same time.

To do this, all you have to do is to add a multiprocess parameter to pipeline\_run:

```
>>> pipeline_run([second_task], multiprocess = 5)
```

In this case, ruffus will try to run up to 5 jobs at the same time. Since our second task only has three jobs, these will be started simultaneously.

## 5.26.6 Up-to-date jobs are not re-run unnecessarily

A job will be run only if the output file timestamps are out of date. If you ran the same code a second time,

```
>>> pipeline_run([second_task])
```

**Nothing would happen because:**

- job1.output2 is more recent than job1.output1 and
- job2.output2 is more recent than job2.output1 and
- job3.output2 is more recent than job3.output1.

**Let us see what happens when just 1 out of 3 pieces of data is modified**

```
open("job1.input1", "w")
pipeline_run([second_task], verbose = 2, multiprocess = 5)
```

You would see that only the out of date jobs (highlighted) have been re-run:

```
>>> pipeline_run([second_task], verbose = 2, multiprocess = 5)
Job = [job1.input -> job1.output1, some_extra.string.for_example, 14] completed
Job = [job3.input -> job3.output1, some_extra.string.for_example, 14] unnecessary: already up to date
Job = [job2.input -> job2.output1, some_extra.string.for_example, 14] unnecessary: already up to date
Completed Task = first_task
Job = [job1.output1 -> job1.output2] completed
Job = [job2.output1 -> job2.output2] unnecessary: already up to date
Job = [job3.output1 -> job3.output2] unnecessary: already up to date
Completed Task = second_task
```

## 5.26.7 Intermediate files

In the above examples, the *input* and *output* parameters are file names. Ruffus was designed for pipelines which save intermediate data in files. This is not compulsory but saving your data in files at each step provides a few advantages:

1. Ruffus can use file system time stamps to check if your pipeline is up to date
2. Your data is persistent across runs
3. This is a good way to pass large amounts of data across processes and computational nodes

Otherwise, task parameters could be all sorts of data, from lists of files, to numbers, sets or tuples. Ruffus imposes few constraints on what *you* would like to send to each stage of your pipeline.

**Ruffus** does, however, assume that all strings in your *input* and *output* parameters represent file names.

*input* parameters which contains a *glob* pattern (e.g. `*.txt`) are expanded to the matching file names.

### 5.26.8 @transform is a 1 to 1 operation

`@transform` is a 1:1 operation because it keeps the number of jobs constant entering and leaving the task. Each job can accept, for example, a pair of files as its input, or generate more than one output files.

Let us see this in action using the previous example:

- `first_task_params` is changed to 3 *pairs* of file names
- **@transform for `first_task` is modified to produce *pairs* of file names**

- `.output.1`
- `.output.extra.1`

```
from ruffus import *

#-----
#   Create pairs of input files
#
first_task_params = [
    ['job1.a.input', 'job1.b.input'],
    ['job2.a.input', 'job2.b.input'],
    ['job3.a.input', 'job3.b.input'],
]

for input_file_pairs in first_task_params:
    for input_file in input_file_pairs:
        open(input_file, "w")

#-----
#   first task
#
@transform(first_task_params, suffix(".input"),
           [".output.1",
            ".output.extra.1"],
           "some_extra.string.for_example", 14)
def first_task(input_files, output_file_pairs,
               extra_parameter_str, extra_parameter_num):
    # make both pairs of output files
    for output_file in output_file_pairs:
        open(output_file, "w")

#-----
#   second task
#
@transform(first_task, suffix(".output.1"), ".output2")
def second_task(input_files, output_file):
    # make output file
    open(output_file, "w")

#-----
```

```
#           Run
#
# pipeline_run([second_task])
```

This gives the following results:

```
>>> pipeline_run([pipeline_task])
```

We see that apart from having a file pair where previously there was a single file, little else has changed. We still have three pieces of data going through the pipeline in three parallel jobs.

## 5.27 Step 3: Understanding how your pipeline works

- *Simple tutorial overview*
- *pipeline functions* in detail

---

**Note:** Remember to look at the example code:

- *Python Code for step 3*
- 

The trickiest part of developing pipelines is understanding how your data flows through the pipeline.

Parameters and files are passed from one task to another down the chain of pipelined functions.

Whether you are learning how to use **ruffus**, or trying out a new feature in **ruffus**, or just have a horrendously complicated pipeline to debug (we have colleagues with >100 criss-crossing pipelined stages), your best friend is *pipeline\_printout(...)*

### 5.27.1 Printing out which jobs will be run

*pipeline\_printout(...)* takes the same parameters as *pipeline\_run* but just prints the tasks which are and are not up-to-date.

The *verbose* parameter controls how much detail is displayed.

Let us take the two step *pipelined code* we have previously written, but call *pipeline\_printout(...)* instead of *pipeline\_run(...)*. This lists the tasks which will be run in the pipeline:

```
>>> pipeline_printout(sys.stdout, [second_task])
Tasks which will be run:
Task = first_task
Task = second_task } Tasks which will be run
```

To see the input and output parameters of each job in the pipeline, we can increase the verbosity from the default (1) to 3:

```
>>> pipeline_printout(sys.stdout, [second_task], verbose = 3 )
```

---

Tasks which will be run:

```
Task = first_task
    Job = [None
           ->job1.stage1] ← Job parameters
        Job needs update: Missing file job1.stage1
    Job = [None
           ->job2.stage1] ← Job parameters
        Job needs update: Missing file job2.stage1

Task = second_task
    Job = [job1.stage1
           ->job1.stage2,      1st_job] ← Job parameters
        Job needs update: Missing file job1.stage1
    Job = [job2.stage1
           ->job2.stage2,      2nd job] ← Job parameters
        Job needs update: Missing file job2.stage1
```

---

This is very useful for checking that the input and output parameters have been specified correctly.

### 5.27.2 Determining which jobs are out-of-date or not

It is often useful to see which tasks are or are not up-to-date. For example, if we were to run the pipeline in full, and then modify one of the intermediate files, the pipeline would be partially out of date.

Let us start by run the pipeline in full but then modify `job1.stage1` so that the second task is no longer up-to-date:

```
pipeline_run([second_task])

# modify job1.stage1
open("job1.stage1", "w").close()
```

At a verbosity of 5, even jobs which are up-to-date will be displayed. We can now see that the there is only one job in `second_task(...)` which needs to be re-run because `job1.stage1` has been modified after `job1.stage2` (highlighted in blue):

```
>>> pipeline_printout(sys.stdout, [second_task], verbose = 5)

Tasks which are up-to-date:
Task = first_task
    Job = [None
           ->job1.stage1]
    Job up-to-date
    Job = [None
           ->job2.stage1]
    Job up-to-date

Tasks which will be run:
Task = second_task
    Job = [job1.stage1
           ->job1.stage2,      1st_job]
        Job needs update:
            Need update file times= [
                [(1269025787.0, 'job1.stage1')],
                [(1269025785.0, 'job1.stage2')] ]
    Job = [job2.stage1
           ->job2.stage2,      2nd_job]
    Job up-to-date
```

Out of date job with the  
file times of the relevant  
input/output files displayed

## 5.28 Step 4: Displaying the pipeline visually

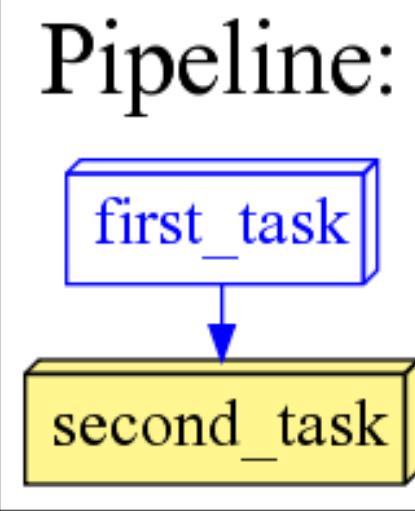
- *Simple tutorial overview*
- *pipeline functions in detail*

**Note:** Remember to look at the example code:

- *Python Code for step 4*

### 5.28.1 Printing out a flowchart of our pipeline

It is all very well being able to trace the data flow through the pipeline. Sometimes, however, we need a bit of eye-candy.



We can see this flowchart of our fledgling pipeline by executing:

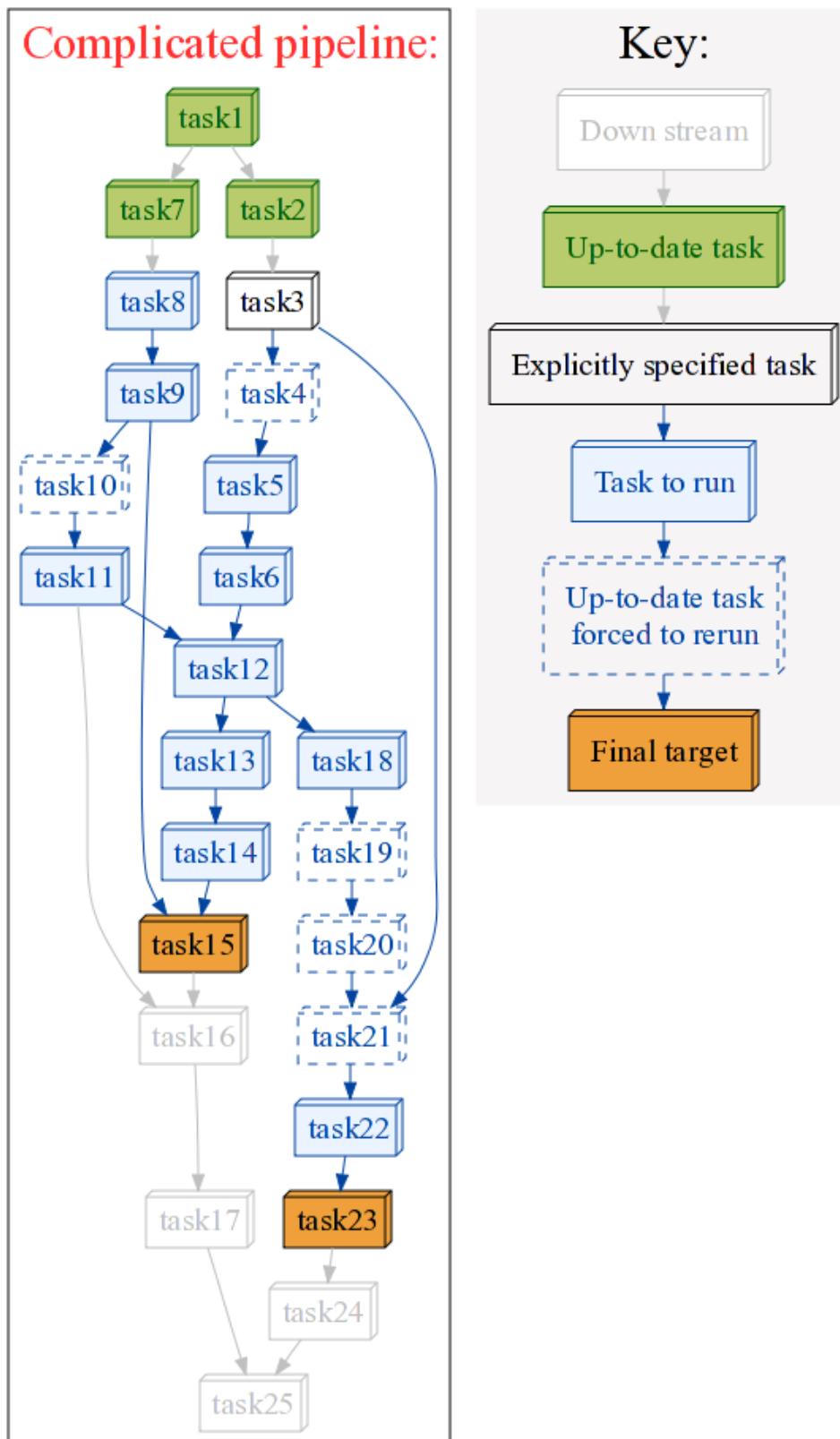
```
pipeline_printout_graph ( 'flowchart.svg', 'svg', [second_task], no_key_legend)
```

Flowcharts can be printed in a large number of formats including jpg, svg, png and pdf provided that the dot programme from Graphviz is installed.

For this simple case, we have omitted the legend key which distinguishes between the different states of the various tasks. (See below for the legend key.)

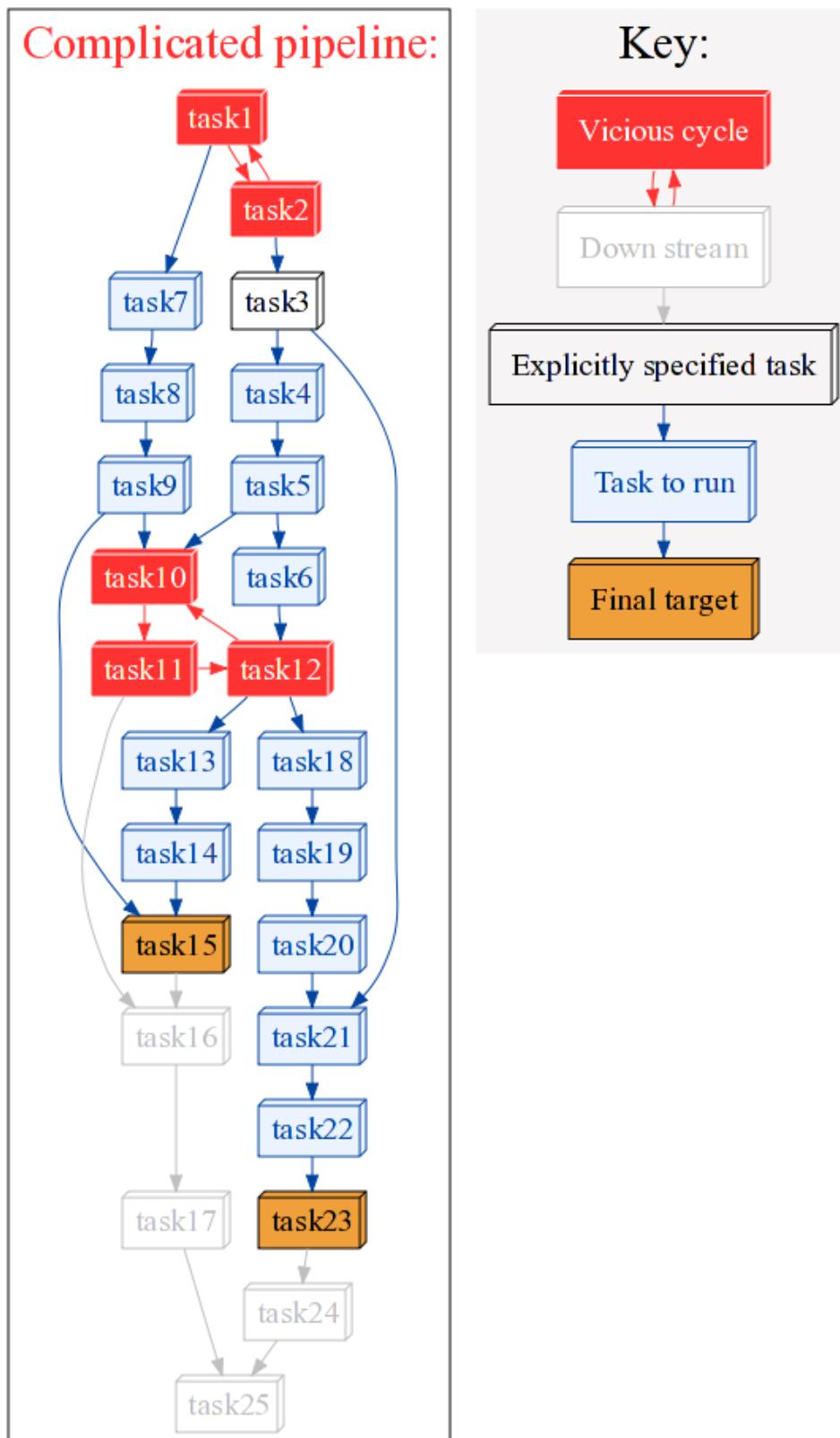
### 5.28.2 Horribly complicated pipelines!

Flowcharts are especially useful if you have really complicated pipelines, such as



### 5.28.3 Circular dependency errors in pipelines!

Especially, if the pipeline is not set up properly, and vicious circular dependencies are present:



## 5.29 Step 5: Splitting up large tasks / files

- Simple tutorial overview
- @split in detail

**Note:** Remember to look at the example code:

- Python Code for step 5

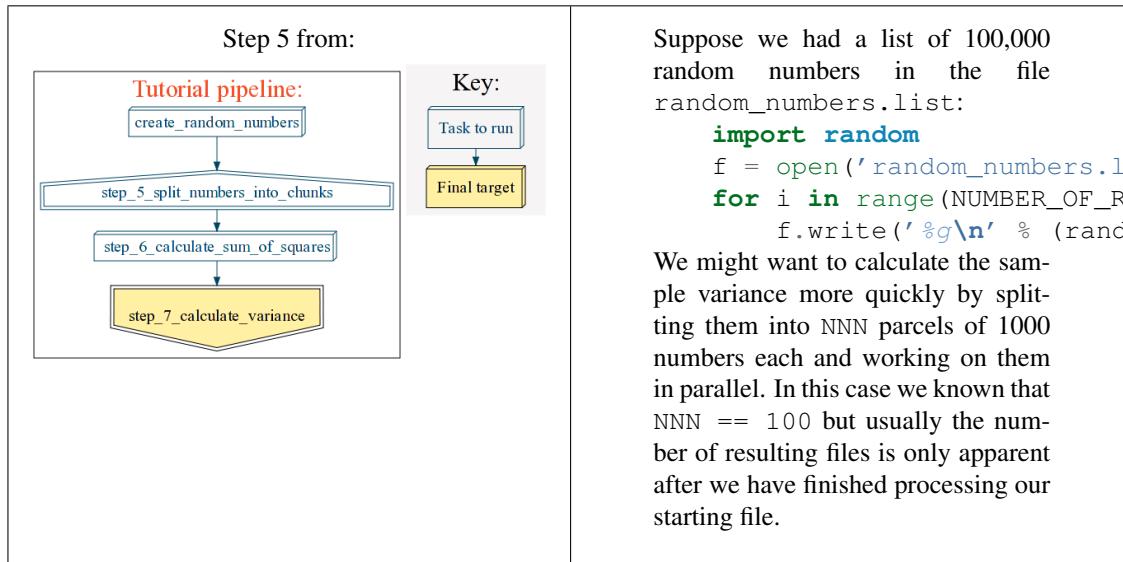
The second half of this tutorial is a worked example to calculate the sample variance of 10,000 random numbers.

This is similar to many computational projects: we are tackling a big problem by splitting it up into many tiny problems solved in parallel. We can then merge our piecemeal solutions into our final answer. These [embarassingly parallel](#) problems motivated the original design of **Ruffus**.

**Ruffus** has three dedicated decorators to handle these problems with ease:

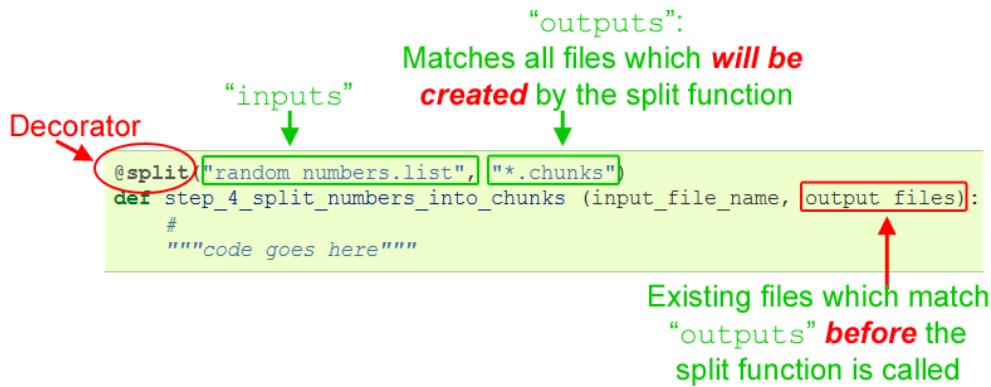
- @split to break up the big problem
- @transform to solve the parts in parallel
- @merge to merge our piecemeal solutions into the final answer.

### 5.29.1 Splitting up a long list of random numbers to calculate their variance



Our pipeline function needs to take the random numbers file `random_numbers.list`, read the random numbers from it, and write to a new file every 100 lines.

The *Ruffus* decorator `@split` is designed specifically for splitting up input into an indeterminate NNN number of output files:



Ruffus will set

```

input_file_name to "random_numbers.list"
output_files to all files which match *.chunks (i.e. "1.chunks", "2.chunks"
etc.).

```

The first time you run this function `*.chunks` will return an empty list because no `.chunks` files have been created, resulting in the following:

```
step_5_split_numbers_into_chunks ("random_numbers.list", [])
```

After that `*.chunks` will match the list of current `.chunks` files created by the previous pipeline run. Some of these files will be out of date or superfluous. These file names are usually only useful for removing detritus from previous runs (have a look at `step_5_split_numbers_into_chunks(...)`).

---

**Note:** The great value of specifying correctly the list of *output* files will become apparent in the next step of this tutorial when we shall see how pipeline tasks can be “chained” together conveniently.

Remember to specify `glob`s patterns which match *all* the files you are splitting up. You can cover different directories, or groups of file names by using a list of `glob`s: e.g.

```

@split("input.file", ['a*.bits', 'b*.pieces', 'somewhere_else/c*.stuff'])
def split_function (input_filename, output_files):
    "Code to split up 'input.file'"

```

---

## 5.30 Step 6: Running jobs in parallel

- Simple tutorial overview
- `@transform` in detail

---

**Note:** Remember to look at the example code:

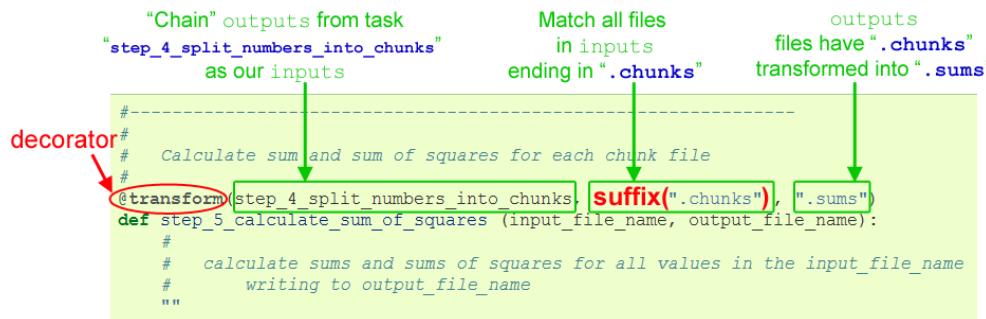
- Python Code for step 6
- 

### 5.30.1 Calculating sums and sum of squares in parallel

Now that we have many smaller lists of numbers in separate files, we can calculate their sums and sum of squares in parallel.

All we need is a function which takes a \*.chunk file, reads the numbers, calculates the answers and writes them back out to a corresponding \*.sums file.

*Ruffus* magically takes care of applying this task function to all the different data files in parallel.



The first thing to note about this example is that the *input* files are not specified as a *glob* (e.g. `*.chunk`) but as the preceding task.

*Ruffus* will take all the files produced by `step_5_split_numbers_into_chunks()` and feed them as the *input* into step 6.

This handy shortcut also means that **Ruffus** knows that `step_6_calculate_sum_of_squares` depends on `step_5_split_numbers_into_chunks` and an additional `@follows` directive is unnecessary.

The use of `suffix` within the decorator tells *Ruffus* to take all *input* files with the `.chunks` suffix and substitute a `.sums` suffix to generate the corresponding *output* file name.

Thus if `step_5_split_numbers_into_chunks` created

```

"1.chunks"
"2.chunks"
"3.chunks"

```

This would result in the following function calls:

```

step_6_calculate_sum_of_squares ("1.chunk", "1.sum")
step_6_calculate_sum_of_squares ("2.chunk", "2.sum")
step_6_calculate_sum_of_squares ("3.chunk", "3.sum")

# etc...

```

---

**Note:** It is possible to generate *output* filenames using more powerful regular expressions as well. See the `@transform` syntax documentation for more details.

---

## 5.31 Step 7: Merging results back together

- Simple tutorial overview
- @merge in detail

---

**Note:** Remember to look at the example code:

- Python Code for step 7

Now that we have all the partial solutions in `*.sums`, we can merge them together to generate the final answer: the variance of all 100,000 random numbers.

### 5.31.1 Calculating variances from the sums and sum of squares of all chunks

If we add up all the sums, and sum of squares we calculated previously, we can obtain the variance as follows:

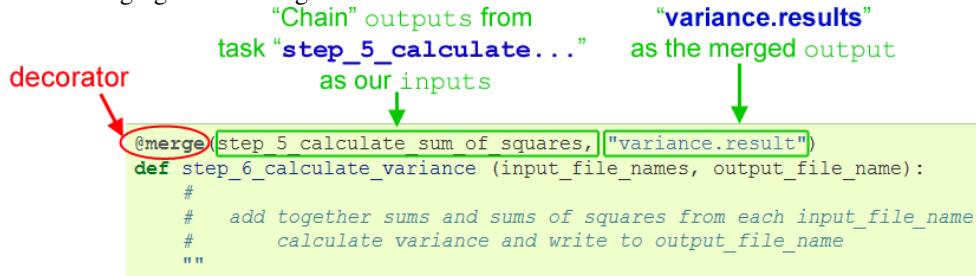
```
variance = (sum_squared - sum * sum / N) / N
```

where N is the number of values

See the [wikipedia](#) entry for a discussion of why this is a very naive approach!

To do this, all we have to do is merge together all the values in `*.sums`, i.e. add up the sums and `sum_squared` for each chunk. We can then apply the above (naive) formula.

Merging files is straightforward in **Ruffus**:

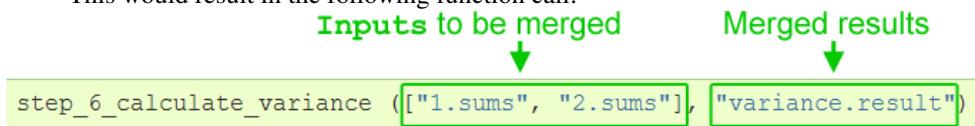


The `@merge` decorator tells *Ruffus* to take all the files from the step 6 task (i.e. `*.sums`), and produced a merged file in the form of "variance.result".

Thus if `step_6_calculate_sum_of_squares` created

1. `.sums` and
2. `.sums` etc.

This would result in the following function call:



The final result is, of course, in "variance.result".

## 5.32 Step 8: Signal the completion of each stage of our pipeline

- Simple tutorial overview
- `@posttask` in detail

**Note:** Remember to look at the example code:

- Python Code for step 8
- 

Let us finish by celebrating the success of our modest pipeline example.

### 5.32.1 Running some code to show that a stage of the pipeline has finished

A common requirement is to take some extra action when a particular *task* or stage of a pipeline is complete.

This can range from printing out some message, or touching some sentinel file, to emailing the author.

This is particularly useful if the *task* is a recipe apply to an unspecified number of parameters in parallel in different *jobs*

The “extra action” can be added to a *Ruffus* pipeline using the `@posttask` decorator.

Let us print a “hooray” message to show that we have finished calculating variances.

The diagram shows a code snippet with annotations. A red arrow points from the word "Decorator" to the `@posttask` decorator in the first line. A green arrow points from the text "Function to run when" to the annotation "all jobs are complete" above the code. The code itself is:

```
def print_hooray():
    sys.stdout.write("hooray\n")
@posttask(print_hooray)
@merge(step_5_calculate_sum_of_squares, "variance.result")
def step_6_calculate_variance (input_file_names, output_file_name):
    ""
```

This is such a short function, we can even write it in-line:

```
@posttask(lambda: sys.stdout.write("hooray\n"))
@merge(step_6_calculate_sum_of_squares, "variance.result")
def step_7_calculate_variance (input_file_names, output_file_name):
    ""
```

### 5.32.2 Touching a sentinel file after finishing a pipeline stage

Very often we would like to mark the completion of a pipeline stage by using the date/time stamp of a “sentinel” file.

This is such a common requirement that *Ruffus* even has special syntax for this in the form of `touch_file`.

```
@posttask(touch_file("sentinel_flag"))
def your_pipeline_function (input_file_names, output_file_name):
    ""
```

The file `sentinel_flag` will be created (if it did not exist) or its date/time stamp changed to the current time whenever this stage of the pipeline is completed.

### 5.32.3 Adding several post task actions

You can, of course, add more than one different action to be taken on completion of the task, either by stacking up `@posttask` decorators or by including several functions in the same `@posttask`:

```
@posttask(print_hooray, print_whoopee)
@posttask(touch_file("sentinel_flag"))
def your_pipeline_function (input_file_names, output_file_name):
    ""
```

### 5.32.4 Finding out more about Ruffus

This wraps up our short tutorial on the **Ruffus**.

Here are a few useful topics you may be interested in:

- *How to summarise disparate input by category*
- *How to log pipeline progress*
- *How exceptions are handled*

To find out more about **Ruffus**, you can read the *manual* or just start using **Ruffus**.

Email the authors at ruffus\_lib at llew.org.uk if you have any comments or suggestions.

Happy pipelining!

Old Manual Code:

## 5.33 Code for Chapter 9: Checking dependencies to run tasks in order

- *Manual overview*
- *Back*

This example shows how dependencies work

### 5.33.1 Code

```
from ruffus import *
import json

import time
def task_helper(infile, outfile):
    """
    cat input file content to output file
    after writing out job parameters
    """
    if infile:
        output_text = "".join(sorted(open(infile).readlines()))
    else:
        output_text = "None"
    output_text += json.dumps(infile) + " -> " + json.dumps(outfile) + "\n"
    open(outfile, "w").write(output_text)

    #
    #      task1
    #
@files(None, 'a.1')
def task1(infile, outfile):
    """
    First task
    """
    task_helper(infile, outfile)
```

```
#  
#      task2  
#  
@transform(task1, regex(r'.1'), '.2')  
def task2(infile, outfile):  
    """  
    Second task  
    """  
    task_helper(infile, outfile)  
  
#  
#      task3  
#  
@transform(task2, regex(r'.2'), '.3')  
def task3(infile, outfile):  
    """  
    Third task  
    """  
    task_helper(infile, outfile)  
  
#  
#      task4  
#  
@transform(task3, regex(r'.3'), '.4')  
def task4(infile, outfile):  
    """  
    Fourth task  
    """  
    task_helper(infile, outfile)  
  
pipeline_printout_graph ("flowchart.png", "png", [task4], draw_vertically = True, no_key_legend  
pipeline_run([task4])
```

### 5.33.2 Resulting Output

```
>>> pipeline_run([task4], multiprocess = 10, logger = logger_proxy)  
job = [null, "a.1"]  
job = ["a.1", "a.2"]  
job = ["a.2", "a.3"]  
job = ["a.3", "a.4"]
```

## 5.34 Code for Chapter 16: Logging progress through a pipeline

- *Manual overview*
- *Back*

This example shows how to log messages from within each of your pipelined functions.

### 5.34.1 Code



```

@transform(task2, regex('.2'), '.3', logger_proxy, logging_mutex)
def task3(infiles, outfiles, *extra_params):
    """
    Third task
    """
    test_job_io(infiles, outfiles, extra_params)

#
#      task4
#
@transform(task3, regex('.3'), '.4', logger_proxy, logging_mutex)
def task4(infiles, outfiles, *extra_params):
    """
    Fourth task
    """
    test_job_io(infiles, outfiles, extra_params)

#
# Necessary to protect the "entry point" of the program under windows.
#     see: http://docs.python.org/library/multiprocessing.html#multiprocessing-programming
#
pipeline_run([task4], multiprocess = 10, logger = logger_proxy)

```

## 5.34.2 Resulting Output

```

>>> pipeline_run([task4], multiprocess = 10, logger = logger_proxy)
job = [null, "a.1"]
job = ["a.1", "a.2"]
job = ["a.2", "a.3"]
job = ["a.3", "a.4"]

```

Pipeline.log will contain our unimaginative log messages:

```

2009-11-15 03:04:55,884 - my_logger - DEBUG - job = [null, "a.1"], process name = PoolWorker-2
2009-11-15 03:04:56,941 - my_logger - INFO - Job = [None -> a.1, <LoggingProxy>, <thread.1>]
2009-11-15 03:04:56,942 - my_logger - INFO - Completed Task = task1
2009-11-15 03:04:56,945 - my_logger - DEBUG - job = ["a.1", "a.2"], process name = PoolWorker-4
2009-11-15 03:04:57,962 - my_logger - INFO - Job = [a.1 -> a.2, <LoggingProxy>, <thread.1>]
2009-11-15 03:04:57,962 - my_logger - INFO - Completed Task = task2
2009-11-15 03:04:57,965 - my_logger - DEBUG - job = ["a.2", "a.3"], process name = PoolWorker-3
2009-11-15 03:04:59,009 - my_logger - INFO - Job = [a.2 -> a.3, <LoggingProxy>, <thread.1>]
2009-11-15 03:04:59,010 - my_logger - INFO - Completed Task = task3
2009-11-15 03:04:59,013 - my_logger - DEBUG - job = ["a.3", "a.4"], process name = PoolWorker-5
2009-11-15 03:05:00,024 - my_logger - INFO - Job = [a.3 -> a.4, <LoggingProxy>, <thread.1>]
2009-11-15 03:05:00,025 - my_logger - INFO - Completed Task = task4

```

## 5.35 Code for Chapter 10: Generating parameters on the fly

- *Manual overview*
- *@files on-the-fly syntax in detail*

- *Back*

This script takes N pairs of input file pairs (with the suffixes .gene and .gwas) and runs them against M sets of simulation data (with the suffix .simulation). A summary per input file pair is then produced

In pseudo-code:

#### **STEP\_1:**

```
for n_file in NNN_pairs_of_input_files:  
    for m_file in MMM_simulation_data:  
  
        [n_file.gene,  
         n_file.gwas,  
         m_file.simulation] -> n_file.m_file.simulation_res
```

### STEP 2:

```
for n_file in NNN_pairs_of_input_files:  
    n_file.*.simulation_res -> n_file.mean
```

n = CNT\_GENE\_GWAS\_FILES  
m = CNT\_SIMULATION\_FILES

### 5.35.1 Code



```
#_____
#
#      setup_simulation_data
#
#_____
#
#  mkdir: makes sure output directories exist before task
#
@follows (mkdir(gene_data_dir, simulation_data_dir))
def setup_simulation () :
    """
    create simulation files
    """

    for i in range(CNT_GENE_GWAS_FILES):
        open(os.path.join(gene_data_dir, "%03d.gene" % i), "w")
        open(os.path.join(gene_data_dir, "%03d.gwas" % i), "w")
    #
    # gene files without corresponding gwas and vice versa
    open(os.path.join(gene_data_dir, "orphan1.gene"), "w")
    open(os.path.join(gene_data_dir, "orphan2.gwas"), "w")
    open(os.path.join(gene_data_dir, "orphan3.gwas"), "w")
    #
    for i in range(CNT_SIMULATION_FILES):
        open(os.path.join(simulation_data_dir, "%03d.simulation" % i), "w")

#_____
#
#      cleanup_simulation_data
#
#_____
#
def try_rmdir (d):
    if os.path.exists(d):
        try:
            os.rmdir(d)
        except OSError:
            sys.stderr.write("Warning:\t%s is not empty and will not be removed.\n" % d)

def cleanup_simulation_data () :
    """
    cleanup files
    """
    sys.stderr.write("Cleanup working directory and simulation files.\n")
    #
    #  cleanup gene and gwas files
    #
    for f in glob.glob(os.path.join(gene_data_dir, "*.gene")):
        os.unlink(f)
    for f in glob.glob(os.path.join(gene_data_dir, "*.gwas")):
        os.unlink(f)
```

```

try_rmdir(gene_data_dir)
#
#   cleanup simulation
#
for f in glob.glob(os.path.join(simulation_data_dir, "*.simulation")):
    os.unlink(f)
try_rmdir(simulation_data_dir)
#
#   cleanup working_dir
#
for f in glob.glob(os.path.join(working_dir, "simulation_results", "*.simulation_res")):
    os.unlink(f)
try_rmdir(os.path.join(working_dir, "simulation_results"))
#
for f in glob.glob(os.path.join(working_dir, "*.mean")):
    os.unlink(f)
try_rmdir(working_dir)

#
# Step 1:
#
#       for n_file in NNN_pairs_of_input_files:
#           for m_file in MMM_simulation_data:
#
#               [n_file.gene,
#                n_file.gwas,
#                m_file.simulation] -> working_dir/n_file.m_file.simulation_res
#
#
def generate_simulation_params():
    """
    Custom function to generate
    file names for gene/gwas simulation study
    """
    simulation_files, simulation_file_roots = get_simulation_files()
    gene_gwas_file_pairs, gene_gwas_file_roots = get_gene_gwas_file_pairs()
    #
    for sim_file, sim_file_root in izip(simulation_files, simulation_file_roots):
        for (gene, gwas), gene_file_root in izip(gene_gwas_file_pairs, gene_gwas_file_roots):
            #
            result_file = "%s.%s.simulation_res" % (gene_file_root, sim_file_root)
            result_file_path = os.path.join(working_dir, "simulation_results", result_file)
            #
            yield [gene, gwas, sim_file], result_file_path, gene_file_root, sim_file_root, result_file

    #
    # mkdir: makes sure output directories exist before task
    #
@follows(mkdir(working_dir, os.path.join(working_dir, "simulation_results")))
@files(generate_simulation_params)
def gwas_simulation(input_files, result_file_path, gene_file_root, sim_file_root, result_file):
    """
    Dummy calculation of gene gwas vs simulation data
    Normally runs in parallel on a computational cluster

```

```
"""
(gene_file,
gwas_file,
simulation_data_file) = input_files
#
simulation_res_file = open(result_file_path, "w")
simulation_res_file.write("%s + %s -> %s\n" % (gene_file_root, sim_file_root, result_file))

#_____
#
# Step 2:
#
#     Statistical summary per gene/gwas file pair
#
#     for n_file in NNN_pairs_of_input_files:
#         working_dir/simulation_results/n.*.simulation_res
#             -> working_dir/n.mean
#
#_____
#_____
```

```
@collate(gwas_simulation, regex(r"simulation_results/(\d+)\.\d+.simulation_res"), r"\1.mean")
@posttask(lambda : sys.stdout.write("\nOK\n"))
def statistical_summary (result_files, summary_file):
    """
    Simulate statistical summary
    """
    summary_file = open(summary_file, "w")
    for f in result_files:
        summary_file.write(open(f).read())
    pipeline_run([setup_simulation_data], multiprocess = 5, verbose = 2)
    pipeline_run([statistical_summary], multiprocess = 5, verbose = 2)

    # uncomment to printout flowchar
    #
    # pipeline_printout(sys.stdout, [statistical_summary], verbose=2)
    # graph_printout ("flowchart.jpg", "jpg", [statistical_summary])
    #

    cleanup_simulation_data ()
```

## 5.35.2 Resulting Output

```
>>> pipeline_run([setup_simulation_data], multiprocess = 5, verbose = 2)
    Make directories [temp_NxM/gene, temp_NxM/simulation] completed
Completed Task = setup_simulation_data_mkdir_1
    Job completed
Completed Task = setup_simulation_data
```

```
>>> pipeline_run([statistical_summary], multiprocess = 5, verbose = 2)
    Make directories [temp_NxM, temp_NxM/simulation_results] completed
Completed Task = gwas_simulation_mkdir_1
```

```

Job = [[temp_NxM/gene/001.gene, temp_NxM/gene/001.gwas, temp_NxM/simulation/000.simulation]
Job = [[temp_NxM/gene/000.gene, temp_NxM/gene/000.gwas, temp_NxM/simulation/000.simulation]
Job = [[temp_NxM/gene/001.gene, temp_NxM/gene/001.gwas, temp_NxM/simulation/001.simulation]
Job = [[temp_NxM/gene/000.gene, temp_NxM/gene/000.gwas, temp_NxM/simulation/001.simulation]
Job = [[temp_NxM/gene/000.gene, temp_NxM/gene/000.gwas, temp_NxM/simulation/002.simulation]
Job = [[temp_NxM/gene/001.gene, temp_NxM/gene/001.gwas, temp_NxM/simulation/002.simulation]
Completed Task = gwas_simulation
Job = [[temp_NxM/simulation_results/000.000.simulation_res, temp_NxM/simulation_results/000.
Job = [[temp_NxM/simulation_results/001.000.simulation_res, temp_NxM/simulation_results/001.

```

## 5.36 Code for Chapter 6: Applying the same recipe to create many different files

- *Manual overview*
- *@transform syntax in detail*
- *Back*

Our example starts off with data file for different zoo animals.

We are only interested in mammals, and we would like the files of each species to end up in its own directory after processing.

### 5.36.1 Code

```

#
# Start with species files
#
open("mammals.tiger.wild.animals"      , "w")
open("mammals.lion.wild.animals"        , "w")
open("mammals.lion.handreared.animals", "w")
open("mammals.dog.tame.animals"         , "w")
open("mammals.dog.wild.animals"         , "w")
open("reptiles.crocodile.wild.animals", "w")

#
# create destinations for each species
#
import os
for s in ("tiger", "lion", "dog"):
    if not os.path.exists(s):
        os.mkdir(s)

#
# Now summarise files in directories organised by species
#
from ruffus import *
@transform('*.*.animals',
          regex(r'mammals\.(.+)\.(.+)\.animals'), # save species and wild/tame
          r'\1\1.\2.in_my_zoo',                      # same species go together
          r'\1')                                     # extra species name

```

```
def capture_mammals(infile, outfile, species):
    open(outfile, "w").write(open(infile).read() + "\ncaptured %s\n" % species)

pipeline_run([capture_mammals])
```

## 5.36.2 Resulting Output

```
>>> pipeline_run([capture_mammals])
Job = [mammals.dog.tame.animals      -> dog/dog.tame.in_my_zoo, dog] completed
Job = [mammals.dog.wild.animals      -> dog/dog.wild.in_my_zoo, dog] completed
Job = [mammals.lion.handreared.animals -> lion/lion.handreared.in_my_zoo, lion] completed
Job = [mammals.lion.wild.animals      -> lion/lion.wild.in_my_zoo, lion] completed
Job = [mammals.tiger.wild.animals     -> tiger/tiger.wild.in_my_zoo, tiger] completed
Completed Task = capture_mammals
```

Old Tutorial Code:

## 5.37 Code for Step 2: Passing parameters to the pipeline

- *Up*
- *Back*
- @*transform* syntax in detail

### 5.37.1 Code

```
from ruffus import *

#-----
#   Create input files
#
first_task_params = [
    'job1.input',
    'job2.input',
    'job3.input'
]

for input_file in first_task_params:
    open(input_file, "w")

#-----
#
#   first task
#
@transform(first_task_params, suffix(".input"), ".output1",
          "some_extra.string.for_example", 14)
def first_task(input_file, output_file,
               extra_parameter_str, extra_parameter_num):
    # make output file
    open(output_file, "w")
```

```

#-----
#
#     second task
#
@transform(first_task, suffix(".output1"), ".output2")
def second_task(input_file, output_file):
    # make output file
    open(output_file, "w")

#-----
#
#           Run
#
pipeline_run([second_task])

```

## 5.37.2 Resulting Output

```

>>> pipeline_run([second_task])
Job   = [job1.input -> job1.output1, some_extra.string.for_example, 14] completed
Job   = [job2.input -> job2.output1, some_extra.string.for_example, 14] completed
Job   = [job3.input -> job3.output1, some_extra.string.for_example, 14] completed
Completed Task = first_task
Job   = [job1.output1 -> job1.output2] completed
Job   = [job2.output1 -> job2.output2] completed
Job   = [job3.output1 -> job3.output2] completed
Completed Task = second_task

```

## 5.38 Code for Step 3: Displaying the pipeline visually

- *Simple tutorial overview*
- *pipeline functions* in detail
- *Back to Step 3*

### 5.38.1 Display the initial state of the pipeline

```

from ruffus import *
import sys

#-----
#
#     first task
#
task1_param = [
    [ None, 'job1.stage1'], # 1st job
    [ None, 'job2.stage1'], # 2nd job
]

@files(task1_param)
def first_task(no_input_file, output_file):
    open(output_file, "w")

```

```
#  
# pretend we have worked hard  
  
#-----  
#  
# second task  
#  
task2_param = [  
    [ 'job1.stage1', "job1.stage2", "      1st_job"], # 1st job  
    [ 'job2.stage1', "job2.stage2", "      2nd_job"], # 2nd job  
]  
  
@follows(first_task)  
@files(task2_param)  
def second_task(input_file, output_file, extra_parameter):  
    open(output_file, "w")  
    print extra_parameter  
  
pipeline_printout(sys.stdout, [second_task], verbose = 3)
```

## 5.38.2 Resulting Output

```
>>> pipeline_printout(sys.stdout, [second_task])  
  
Task = first_task  
Job = [None -> job1.stage1]  
Job = [None -> job2.stage1]  
  
Task = second_task  
Job = [job1.stage1 -> job1.stage2,      1st_job]  
Job = [job2.stage1 -> job2.stage2,      2nd_job]
```

## 5.38.3 Display the partially up-to-date pipeline

Run the pipeline, modify `job1.stage` so that the second task is no longer up-to-date and printout the pipeline stage again:

```
pipeline_run([second_task])  
  
# modify job1.stage1  
open("job1.stage1", "w").close()
```

At a verbosity of 5, even jobs which are up-to-date will be displayed:

```
>>> pipeline_printout(sys.stdout, [second_task], verbose = 5)
```

---

Tasks which are up-to-date:

```
Task = first_task  
Job = [None  
      ->job1.stage1]  
Job up-to-date  
Job = [None  
      ->job2.stage1]
```

```
Job up-to-date
```

---

```
Tasks which will be run:
```

```
Task = second_task
    Job = [job1.stage1
           ->job1.stage2,      1st_job]
    Job needs update: Need update file times= [[(1269025787.0, 'job1.stage1')], [(1269025787.0, 'job1.stage2'))]]
Job = [job2.stage1
           ->job2.stage2,      2nd_job]
Job up-to-date
```

---

We can now see that the there is only one job in “second\_task” which needs to be re-run because ‘job1.stage1’ has been modified after ‘job1.stage2’

## 5.39 Code for Step 4: Displaying the pipeline visually

- *Simple tutorial overview*
- *pipeline functions* in detail
- *Back to Step 4*

### 5.39.1 Code

```
from ruffus import *
import time

#-----
#
#   first task
#
task1_param = [
    [ None, 'job1.stage1'], # 1st job
    [ None, 'job2.stage1'], # 2nd job
]

@files(task1_param)
def first_task(no_input_file, output_file):
    open(output_file, "w")

#-----
#
#   second task
#
task2_param = [
    [ 'job1.stage1', "job1.stage2", "      1st_job"], # 1st job
    [ 'job2.stage1', "job2.stage2", "      2nd_job"], # 2nd job
]
```

```

@follows(first_task)
@files(task2_param)
def second_task(input_file, output_file, extra_parameter):
    open(output_file, "w")
    print extra_parameter

#-----
#
#      Show flow chart and tasks before running the pipeline
#
print "Show flow chart and tasks before running the pipeline"
pipeline_printout_graph ( open("flowchart_before.png", "w"),
                           "png",
                           [second_task],
                           no_key_legend=True)

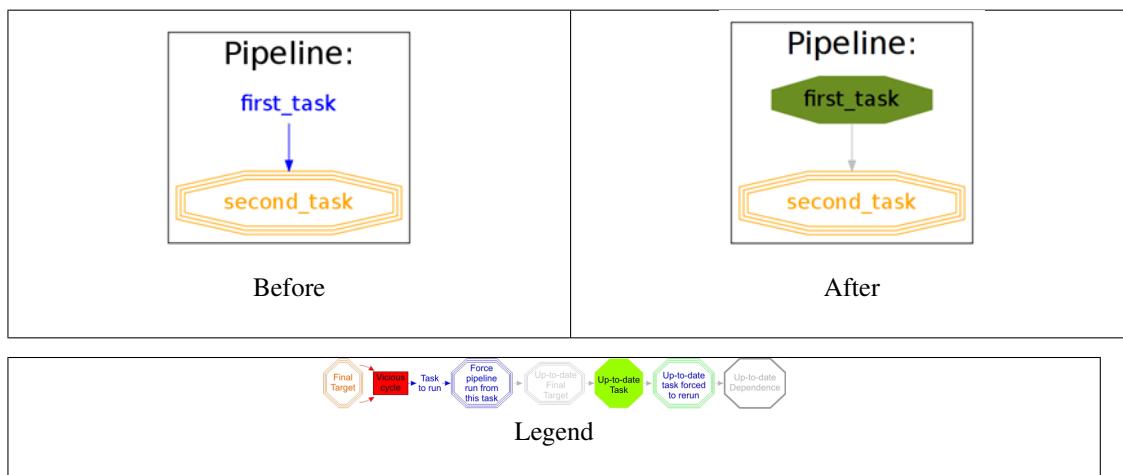
#-----
#
#      Run
#
pipeline_run([second_task])

# modify job1.stage1
open("job1.stage1", "w").close()

#-----
#
#      Show flow chart and tasks after running the pipeline
#
print "Show flow chart and tasks after running the pipeline"
pipeline_printout_graph ( open("flowchart_after.png", "w"),
                           "png",
                           [second_task],
                           no_key_legend=True)

```

### 5.39.2 Resulting Flowcharts



## 5.40 Code for Step 5: Splitting up large tasks / files

- Simple tutorial overview
- @split in detail
- back to step 5

### 5.40.1 Code

```

NUMBER_OF_RANDOMS = 10000
CHUNK_SIZE = 1000

from ruffus import *
import time

import random

#-----
#
#    Create random numbers
#
@files(None, "random_numbers.list")
def create_random_numbers(input_file_name, output_file_name):
    f = open(output_file_name, "w")
    for i in range(NUMBER_OF_RANDOMS):
        f.write("%g\n" % (random.random() * 100.0))

#-----
#
#    Split initial file
#
@follows(create_random_numbers)
@split("random_numbers.list", "*.chunks")
def step_5_split_numbers_into_chunks (input_file_name, output_files):
    """
        Splits random numbers file into XXX files of CHUNK_SIZE each
    """
    #
    #    clean up files from previous runs
    #
    for f in glob.glob("*.chunks"):
        os.unlink(f)
    #
    #    create new file every CHUNK_SIZE lines and
    #        copy each line into current file
    #
    output_file = None
    cnt_files = 0
    for i, line in enumerate(open(input_file_name)):
        if i % CHUNK_SIZE == 0:
            cnt_files += 1
            output_file = open("%d.chunks" % cnt_files, "w")
            output_file.write(line)

    pipeline_run([step_5_split_numbers_into_chunks], verbose = 2)

```

## 5.40.2 Resulting Output

```
>>> pipeline_run([step_5_split_numbers_into_chunks], verbose = 2)

Start Task = create_random_numbers

    Job = [None -> random_numbers.list] Missing file random_numbers.list
    Job = [None -> random_numbers.list] completed
Completed Task = create_random_numbers
Start Task = step_5_split_numbers_into_chunks
Splits random numbers file into XXX files of CHUNK_SIZE each
    Job = [random_numbers.list -> *.chunks] Missing output file
    Job = [random_numbers.list -> *.chunks] completed
Completed Task = step_5_split_numbers_into_chunks
```

## 5.41 Code for Step 6: Running jobs in parallel

- Simple tutorial overview
- @transform in detail
- back to step 6

### 5.41.1 Code

```
NUMBER_OF_RANDOMS = 10000
CHUNK_SIZE = 1000

from ruffus import *
import time

import random
import glob

#-----
#
#      Create random numbers
#
@files(None, "random_numbers.list")
def create_random_numbers(input_file_name, output_file_name):
    f = open(output_file_name, "w")
    for i in range(NUMBER_OF_RANDOMS):
        f.write("%g\n" % (random.random() * 100.0))

#-----
#
#      Split initial file
#
@follows(create_random_numbers)
@split("random_numbers.list", "*.chunks")
def step_5_split_numbers_into_chunks (input_file_name, output_files):
    """
        Splits random numbers file into XXX files of CHUNK_SIZE each
    """
```

```

#
#   clean up files from previous runs
#
for f in glob.glob("*.chunks"):
    os.unlink(f)
#
#
#   create new file every CHUNK_SIZE lines and
#       copy each line into current file
#
output_file = None
cnt_files = 0
for i, line in enumerate(open(input_file_name)):
    if i % CHUNK_SIZE == 0:
        cnt_files += 1
        output_file = open("%d.chunks" % cnt_files, "w")
        output_file.write(line)

#-----
#
#   Calculate sum and sum of squares for each chunk file
#
@transform(step_5_split_numbers_into_chunks, suffix(".chunks"), ".sums")
def step_6_calculate_sum_of_squares (input_file_name, output_file_name):
    output = open(output_file_name, "w")
    sum_squared, sum = [0.0, 0.0]
    cnt_values = 0
    for line in open(input_file_name):
        cnt_values += 1
        val = float(line.rstrip())
        sum_squared += val * val
        sum += val
    output.write("%s\n%s\n%d\n" % (repr(sum_squared), repr(sum), cnt_values))

pipeline_run([step_6_calculate_sum_of_squares], verbose = 1)

```

## 5.41.2 Resulting Output

```

>>> pipeline_run([step_6_calculate_sum_of_squares], verbose = 1)
Job = [None -> random_numbers.list] unnecessary: already up to date
Completed Task = create_random_numbers
Job = [random_numbers.list -> *.chunks] unnecessary: already up to date
Completed Task = step_5_split_numbers_into_chunks
Job = [6.chunks -> 6.sum] completed
Job = [1.chunks -> 1.sum] completed
Job = [4.chunks -> 4.sum] completed
Job = [7.chunks -> 7.sum] completed
Job = [2.chunks -> 2.sum] completed
Job = [9.chunks -> 9.sum] completed
Job = [10.chunks -> 10.sum] completed
Job = [3.chunks -> 3.sum] completed
Job = [5.chunks -> 5.sum] completed
Job = [8.chunks -> 8.sum] completed
Completed Task = step_6_calculate_sum_of_squares

```

## 5.42 Code for Step 7: Merging results back together

- Simple tutorial overview
- @merge in detail
- back to step 7

### 5.42.1 Code

```
NUMBER_OF_RANDOMS = 10000
CHUNK_SIZE = 1000

from ruffus import *
import time

import random
import glob

#-----
# Create random numbers
#
@files(None, "random_numbers.list")
def create_random_numbers(input_file_name, output_file_name):
    f = open(output_file_name, "w")
    for i in range(NUMBER_OF_RANDOMS):
        f.write("%g\n" % (random.random() * 100.0))

#-----
# Split initial file
#
@follows(create_random_numbers)
@splits("random_numbers.list", "*.chunks")
def step_5_split_numbers_into_chunks (input_file_name, output_files):
    """
        Splits random numbers file into XXX files of CHUNK_SIZE each
    """
    #
    # clean up files from previous runs
    #
    for f in glob.glob("*.chunks"):
        os.unlink(f)
    #
    # create new file every CHUNK_SIZE lines and
    #     copy each line into current file
    #
    output_file = None
    cnt_files = 0
    for i, line in enumerate(open(input_file_name)):
        if i % CHUNK_SIZE == 0:
            cnt_files += 1
            output_file = open("%d.chunks" % cnt_files, "w")
            output_file.write(line)
```

```

#-----
#
#   Calculate sum and sum of squares for each chunk file
#
@transform(step_5_split_numbers_into_chunks, suffix(".chunks"), ".sums")
def step_6_calculate_sum_of_squares (input_file_name, output_file_name):
    output = open(output_file_name, "w")
    sum_squared, sum = [0.0, 0.0]
    cnt_values = 0
    for line in open(input_file_name):
        cnt_values += 1
        val = float(line.rstrip())
        sum_squared += val * val
        sum += val
    output.write("%s\n%s\n%d\n" % (repr(sum_squared), repr(sum), cnt_values))

#-----
#
#   Calculate sum and sum of squares for each chunk
#
@merge(step_6_calculate_sum_of_squares, "variance.result")
def step_7_calculate_variance (input_file_names, output_file_name):
    """
    Calculate variance naively
    """
    output = open(output_file_name, "w")
    #
    #   initialise variables
    #
    all_sum_squared = 0.0
    all_sum         = 0.0
    all_cnt_values = 0.0
    #
    #   added up all the sum_squared, and sum and cnt_values from all the chunks
    #
    for input_file_name in input_file_names:
        sum_squared, sum, cnt_values = map(float, open(input_file_name).readlines())
        all_sum_squared += sum_squared
        all_sum         += sum
        all_cnt_values += cnt_values
    all_mean = all_sum / all_cnt_values
    variance = (all_sum_squared - all_sum * all_mean) / (all_cnt_values)
    #
    #   print output
    #
    print >>output, variance

#-----
#
#       Run
#
pipeline_run([step_7_calculate_variance], [create_random_numbers], verbose = 1)

```

### 5.42.2 Resulting Output

```
pipeline_run([step_7_calculate_variance], [create_random_numbers], verbose = 1)
    Job = [None -> random_numbers.list] completed
Completed Task = create_random_numbers
    Job = [random_numbers.list -> *.chunks] completed
Completed Task = step_5_split_numbers_into_chunks
    Job = [6.chunks -> 6.sums] completed
    Job = [1.chunks -> 1.sums] completed
    Job = [4.chunks -> 4.sums] completed
    Job = [7.chunks -> 7.sums] completed
    Job = [2.chunks -> 2.sums] completed
    Job = [9.chunks -> 9.sums] completed
    Job = [10.chunks -> 10.sums] completed
    Job = [3.chunks -> 3.sums] completed
    Job = [5.chunks -> 5.sums] completed
    Job = [8.chunks -> 8.sums] completed
Completed Task = step_6_calculate_sum_of_squares
    Job = [[6.sums, 5.sums, 1.sums, 4.sums, 3.sums, 2.sums, 8.sums, 7.sums, 10.sums, 9.sums] ->
Completed Task = step_7_calculate_variance
```

## 5.43 Code for Step 8: Signal the completion of each stage of our pipeline

- *Simple tutorial overview*
- *@posttask in detail*
- *back to step 8*

### 5.43.1 Code

```
NUMBER_OF_RANDOMS = 10000
CHUNK_SIZE = 1000
working_dir = "tempTutorial8/"
```

```
import time, sys, os
from ruffus import *

import random
import glob

#-----
#
# Create random numbers
#
@follows(mkdir(working_dir))
@files(None, working_dir + "random_numbers.list")
def create_random_numbers(input_file_name, output_file_name):
```

```

f = open(output_file_name, "w")
for i in range(NUMBER_OF_RANDOMS):
    f.write("%g\n" % (random.random() * 100.0))

#-----
#
#   Split initial file
#
@follows(create_random_numbers)
@split(working_dir + "random_numbers.list", working_dir + "*.chunks")
def step_5_split_numbers_into_chunks (input_file_name, output_files):
    """
        Splits random numbers file into XXX files of CHUNK_SIZE each
    """
    #
    #   clean up files from previous runs
    #
    for f in glob.glob("*.chunks"):
        os.unlink(f)
    #
    #   create new file every CHUNK_SIZE lines and
    #       copy each line into current file
    #
    output_file = None
    cnt_files = 0
    for i, line in enumerate(open(input_file_name)):
        if i % CHUNK_SIZE == 0:
            cnt_files += 1
            output_file = open(working_dir + "%d.chunks" % cnt_files, "w")
            output_file.write(line)

    #-----
    #
    #   Calculate sum and sum of squares for each chunk file
    #
@transform(step_5_split_numbers_into_chunks, suffix(".chunks"), ".sums")
def step_6_calculate_sum_of_squares (input_file_name, output_file_name):
    output = open(output_file_name, "w")
    sum_squared, sum = [0.0, 0.0]
    cnt_values = 0
    for line in open(input_file_name):
        cnt_values += 1
        val = float(line.rstrip())
        sum_squared += val * val
        sum += val
    output.write("%s\n%s\n%d\n" % (repr(sum_squared), repr(sum), cnt_values))

def print_hooray_again():
    print "hooray again"

def print_whoppee_again():
    print "whoppee again"

#-----
#
#   Calculate sum and sum of squares for each chunk

```

```
#  
@posttask(lambda: sys.stdout.write("hooray\n"))  
@posttask(print_hooray_again, print_whoppee_again, touch_file("done"))  
@merge(step_6_calculate_sum_of_squares, "variance.result")  
def step_7_calculate_variance (input_file_names, output_file_name):  
    """  
    Calculate variance naively  
    """  
    output = open(output_file_name, "w")  
    #  
    # initialise variables  
    #  
    all_sum_squared = 0.0  
    all_sum = 0.0  
    all_cnt_values = 0.0  
    #  
    # added up all the sum_squared, and sum and cnt_values from all the chunks  
    #  
    for input_file_name in input_file_names:  
        sum_squared, sum, cnt_values = map(float, open(input_file_name).readlines())  
        all_sum_squared += sum_squared  
        all_sum += sum  
        all_cnt_values += cnt_values  
    all_mean = all_sum / all_cnt_values  
    variance = (all_sum_squared - all_sum * all_mean) / (all_cnt_values)  
    #  
    # print output  
    #  
    print >>output, variance  
  
#-----  
#  
#      Run  
#  
pipeline_run([step_7_calculate_variance], verbose = 1)
```

## 5.43.2 Resulting Output

```
>> pipeline_run([step_7_calculate_variance], verbose = 1)  
    Make directories [tempTutorial8/] completed  
Completed Task = create_random_numbers_mkdir_1  
    Job = [None -> tempTutorial8/random_numbers.list] completed  
Completed Task = create_random_numbers  
    Job = [tempTutorial8/random_numbers.list -> tempTutorial8/*.chunks] completed  
Completed Task = step_5_split_numbers_into_chunks  
    Job = [tempTutorial8/1.chunks -> tempTutorial8/1.sums] completed  
    Job = [tempTutorial8/10.chunks -> tempTutorial8/10.sums] completed  
    Job = [tempTutorial8/2.chunks -> tempTutorial8/2.sums] completed  
    Job = [tempTutorial8/3.chunks -> tempTutorial8/3.sums] completed  
    Job = [tempTutorial8/4.chunks -> tempTutorial8/4.sums] completed  
    Job = [tempTutorial8/5.chunks -> tempTutorial8/5.sums] completed  
    Job = [tempTutorial8/6.chunks -> tempTutorial8/6.sums] completed  
    Job = [tempTutorial8/7.chunks -> tempTutorial8/7.sums] completed  
    Job = [tempTutorial8/8.chunks -> tempTutorial8/8.sums] completed  
    Job = [tempTutorial8/9.chunks -> tempTutorial8/9.sums] completed  
Completed Task = step_6_calculate_sum_of_squares
```

```
Job = [[tempTutorial8/1.sums, tempTutorial8/10.sums, tempTutorial8/2.sums, tempTutorial8/3.sums], [tempTutorial8/4.sums, tempTutorial8/5.sums, tempTutorial8/6.sums, tempTutorial8/7.sums], [tempTutorial8/8.sums, tempTutorial8/9.sums, tempTutorial8/10.sums]]  
hooray again  
whoppee again  
hooray  
Completed Task = step_7_calculate_variance
```



---

CHAPTER  
**SIX**

---

## INDICES AND TABLES

- *genindex*
- *modindex*
- *search*



## PYTHON MODULE INDEX

r

ruffus.proxy\_logger, ??