JUDI

Release 0.0.1

May 12, 2019

Getting Started

JUDI simplifies building and executing a software pipeline under different parameter settings by automating an efficient execution of the pipeline across the settings.

Consolidated specification of parameter settings JUDI provides an easy and efficient way to specify all possible settings of the parameters which the pipeline needs to be executed for.

Files and tasks independent from parameter settings For each file/task, a user of JUDI just specifies the parameters the file/task depends upon and then builds the pipeline as if there were no parameters at all. JUDI makes sure there are separate instances of the file/task for each setting of the parameters, creates appropriate association between the file instances and the task instances, and automates an efficient execution of the task instances based on their dependency on other tasks.

Easy plug-and-play By decoupling parameter settings from files and tasks, JUDI enables an easy plug and play of different stages of the pipeline.

CHAPTER 1

Install JUDI

JUDI is implemented as a python package and can be installed using the *pip* installer. JUDI depends on python package *pydoit* whose latest versions work with python 3.4 and above. Hence, it is recommended that python 3.4 or above is already installed on the system. JUDI also depends on *pandas*. JUDI has been tested using python 3.6 on Linux based systems.

Assuming you have Python and pandas already, *install judi*:

```
$ pip install doit
$ pip install judi
```

CHAPTER 2

Build and Execute a Simple Pipeline

Let's consider a simple pipeline working on two input files, `a.txt`:

```
Hello you!
```

and `b.txt`:

```
Hello you,
and your friends!
```

For each of the two files, the first stage of the pipeline computes the number of lines, words and characters and stores in a comma-separated file.

The second stage combines the two comma-separated files into a single comma-separated file with an extra field to indicate the source.

2.1 Build pipeline

The Python code `dodo.py` for building this pipeline using JUDI is:

```
from judi import File, Task, add_param, combine_csvs

add_param([1, 2], 'n')

class GetCounts(Task):
  """Count lines, words and characters in file"""
  inputs = {'inp': File('text', path=['a.txt', 'b.txt'])}
  targets = {'out': File('counts.csv')}
  actions = [("(echo line word char file; wc {}) | sed 's/^ \+//;s/ \+/,/g' > {}", ["
↪$inp", "$out"])]

class CombineCounts(Task):
  """Combine counts"""
```

(continues on next page)

(continued from previous page)

```
  mask = ['n']
  inputs = {'inp': GetCounts.targets['out']}
  targets = {'out': File('result.csv', mask=mask, root='.')}
  actions = [(combine_csvs, ["#inp", "#out"])]
```

2.2 Execute pipeline

The pipeline is executed from command line:

```
$ doit -f dodo.py
.  GetCounts:n~1
.  GetCounts:n~2
.  CombineCounts:
```

The `.` before each pipeline task denotes that the task was computed afresh.

The first stage generates two intermediate count files, `judi_files/n~1/counts.csv` and `./judi_files/n~2/counts.csv`.

```
$ cat judi_files/n~1/counts.csv
line,word,char,file
1,2,11,a.txt
$ cat judi_files/n~2/counts.csv
line,word,char,file
2,5,29,b.txt
```

The second stage consolidates the counts in a file `result.csv`:

```
$ cat result.csv
line,word,char,file,n
1,2,11,a.txt,1
2,5,29,b.txt,2
```

2.3 Re-execute pipeline

Invoking `doit` again gives:

```
$ doit -f dodo.py
-- GetCounts:n~1
-- GetCounts:n~2
-- CombineCounts:
```

where `--` denotes that the pipeline task was not executed.

Now let's update the second input file `b.txt` to:

```
Hello you,
your friends,
and whole world!
```

and execute the pipeline again:

```
$ doit -f dodo.py
.  GetCounts:n~2
-- GetCounts:n~1
.  CombineCounts:
```

This time only the counts for `b.txt` is recomputed, the unaffected part of the pipeline for `a.txt` is not executed.

CHAPTER 3

Parameter Database

The main distinguishing feature of JUDI is the use of parameter database which captures all the different settings of parameters that the pipeline being build can possibly be executed for. JUDI maintains a parameter database as a pandas dataframe. The columns indicate the parameters and the rows the settings of the parameters.

In JUDI, each file and each task is associated with a parameter database. However, most of these databases are either the same or share many common parameters. Hence JUDI maintains a global parameter database and the parameter databases to individual file or task is specified through a list of masked parameters.

The global parameter database is populated using the following function:

`judi.paramdb.`**`add_param`**(*param_info*, *name=None*)
Add a parameter or a group of parameters in the global parameter database

Args: param_info (list/dict/Pandas Series/DataFrame): Information about the parameter or group of parameters. If not already so, param_info is converted to a pandas DataFrame and then it is added to the global parameter database via a Cartesian product.

Kwargs: name (str): Used if param_info is a list and denotes the name of the parameter.

Returns: int. The return code: 0 for success and 1 for error!

Raises: None

The global parameter database can be viewed using the following function:

`judi.paramdb.`**`show_param_db`**()
Print the global parameter database

3.1 Some examples

In the following code snippet, the highlighted lines show examples of adding parameters to the global parameter database. Line 4 adds a parameter W having two possible values: 1,2. Line 6 adds a second parameter X with three possible values: a, b, c. Line 8 jointly adds two parameters Y,Z with three possible values (11,aa), (22,aa), (33,bb).

```
from judi import add_param, show_param_db
import pandas as pd

add_param("1 2".split(), 'W')
show_param_db()
add_param("a b c".split(), 'X')
show_param_db()
add_param(pd.DataFrame({'Y':[11, 22, 33], 'Z':['aa', 'aa', 'bb']}))
show_param_db()
```

The contents of the global parameter database can be seen after each addition as follows:

```
Global param db:
   W
0  1
1  2
Global param db:
   W  X
0  1  a
1  1  b
2  1  c
3  2  a
4  2  b
5  2  c
Global param db:
    W  X   Y   Z
0   1  a  11  aa
1   1  a  22  aa
2   1  a  33  bb
3   1  b  11  aa
4   1  b  22  aa
5   1  b  33  bb
6   1  c  11  aa
7   1  c  22  aa
8   1  c  33  bb
9   2  a  11  aa
10  2  a  22  aa
11  2  a  33  bb
12  2  b  11  aa
13  2  b  22  aa
14  2  b  33  bb
15  2  c  11  aa
16  2  c  22  aa
17  2  c  33  bb
```

CHAPTER 4

JUDI File

A JUDI File is associated with a parameter database and actually represents a collection of physical files, each corresponding to a row in the parameter database.

class judi.**File**(*name*, *param=None*, *mask=None*, *path=None*, *root='./judi_files'*)

A file in JUDI is an object of class File and is instantiated by calling the following constructor

__init__(*name*, *param=None*, *mask=None*, *path=None*, *root='./judi_files'*)

Create a JUDI file instance.

Args: name (str): Name of the JUDI file to be created.

Kwargs: param (ParamDb): Parameter database associated with the JUDI file. If param is empty, the golbal parameter database is taken to be associated with the file.

mask (list of str): The list of parameters that are to be masked from the parameter database.

path: Specification of the physical path of the files associated with the JUDI file that is used to generate a column 'path' in the parameter database of the JUDI file. It can be of the following types: 1) callable: a user provider path generator that is passed to pandas apply function, 2) string or list of strings: actual path(s) to the physical files, and 3) None: JUDI automatically generates a path for each row.

root (str): Top level directory where the physical files associated with the JUDI file are created.

4.1 Some examples

The following code snippet creates a global parameter database with two parameters W and X and then creates a file with a parameter database that masks parameter W in the global parameter database.

```
>>> from judi import add_param, show_param_db, File
>>> add_param("1 2".split(), 'W')
0
>>> add_param("a b c".split(), 'X')
0
>>> show_param_db()
```

(continues on next page)

(continued from previous page)

```
Global param db:
   W  X
0  1  a
1  1  b
2  1  c
3  2  a
4  2  b
5  2  c
>>> f = File('test', mask = ['W'])
Creating new directory ./judi_files/X~a
Creating new directory ./judi_files/X~b
Creating new directory ./judi_files/X~c
>>> show_param_db(f.param)
Param db:
   X  name                    path
0  a  test  ./judi_files/X~a/test
1  b  test  ./judi_files/X~b/test
2  c  test  ./judi_files/X~c/test
```

CHAPTER 5

JUDI Task

A JUDI task is associated with a parameter database and actually represents a collection of DoIt tasks, each corresponding to a row in the parameter database.

A JUDI task is python class inherited from the class `Task`. It should define the following class variables.

Essential class variables:

- inputs: A python dictionary for the JUDI files input to the current task.
- targets: A python dictionary for the JUDI files generated by the current task.
- actions: A list of DoIt actions.

Optional class variables:

- mask: A list of parameters that are masked from the global parameter database for the current task.

5.1 Parameter substitution in actions

In additions to the forms of actions supported in DoIt, JUDI supports the following additional form:

- `(func, args)`: Here func could be a string or a callable and args is a list of arguments. When func is a str, it can have placeholders {} which are replaced by the elements of args. When func is a callable it must have only positional arguments provided through args. An element of args can have special strings which are replaced by values as shown in the following table:

Table 1: Argument substitution rules

arg	case	substituted value
'$x'	'x' is an input/target file	Blank separated list of paths for the instances of JUDI file 'x' applicable to the current JUDI task instance.
'#x'	'x' is an input/target file	Parameter database associated with 'x'
'#x'	'x' is a parameter	Value of parameter 'x'
'##'		Python dictionary containing all parameters and their values

5.2 Some special actions

To help users of JUDI summarize data across parameter settings two examples of actions are given in module `judi.utils`.

- `combine_csvs(pdb_big, pdb_small)`: This function row-binds the CSV files given in the path column of the file parameter database `pdb_big` into a single CSV file whose path is given in the path column of the file parameter database `pdb_small`. The function additionally adds the extra parameter settings from `pdb_big` in the consolidated CSV file.
- `combine_pdfs(pdb_big, pdb_small)`: This function combines the pages from PDF files given in the path column of the file parameter database `pdb_big` into a single PDF file whose path is given in the path column of the file parameter database `pdb_small`.

5.3 Some examples

The following code snippet `dodo.py` creates a global parameter database with two parameters W and X and then creates a task with a parameter database that masks parameter W in the global parameter database. Each of the task instances for parameter X then concatenates the input files for all possible values of W. Using the class variable actions, several parameter substitutions have been demonstrated.

```
from judi import add_param, show_param_db, File, Task
add_param("1 2".split(), 'W')
add_param("a b".split(), 'X')
show_param_db()

class Test(Task):
  mask = ['W']
  inputs = {'foo': File('bar', path=lambda x: ''.join([x['X'], x['W']]) + '.txt')}
  targets = {'zoo': File('combined.txt', mask = mask)}
  actions = [('echo ">>" foo files: {}', ['$foo']),
```

(continues on next page)

(continued from previous page)

```
            ('echo ">>" foo param db:'),
            (show_param_db, ['#foo']),
            ('echo ">>" zoo files: {}', ['$zoo']),
            ('echo ">>" zoo param db:'),
            (show_param_db, ['#zoo']),
            ('echo ">>" param X: {}', ['#X']),
            ('echo ">>" All parameters:'),
            (lambda x: print(x), ['##']),
            ('cat {} > {}', ['$foo', '$zoo'])]
```

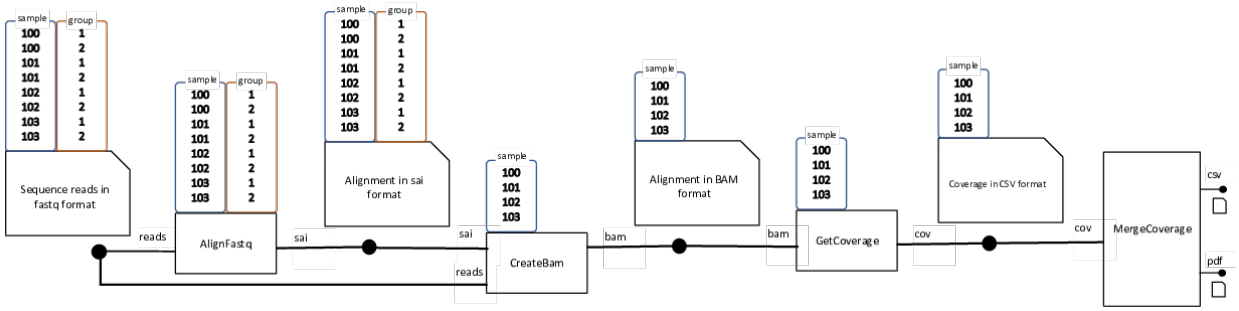
The output of `doit -f dodo.py` is shown below:

```
Global param db:
    W  X
0   1  a
1   1  b
2   2  a
3   2  b
.  Test:X~a
>> foo files: a1.txt a2.txt
>> foo param db:
Param db:
    W name    path
1   1  bar  a1.txt
3   2  bar  a2.txt
>> zoo files: ./judi_files/X~a/combined.txt
>> zoo param db:
Param db:
            name                           path
1  combined.txt  ./judi_files/X~a/combined.txt
>> param X: a
>> All parameters:
X    a
Name: 0, dtype: object
.  Test:X~b
>> foo files: b1.txt b2.txt
>> foo param db:
Param db:
    W name    path
1   1  bar  b1.txt
3   2  bar  b2.txt
>> zoo files: ./judi_files/X~b/combined.txt
>> zoo param db:
Param db:
            name                           path
1  combined.txt  ./judi_files/X~b/combined.txt
>> param X: b
>> All parameters:
X    b
Name: 1, dtype: object
```

CHAPTER 6

Example Pipeline

Let us consider a bioinformatics pipeline with four stages as shown in the following figure:

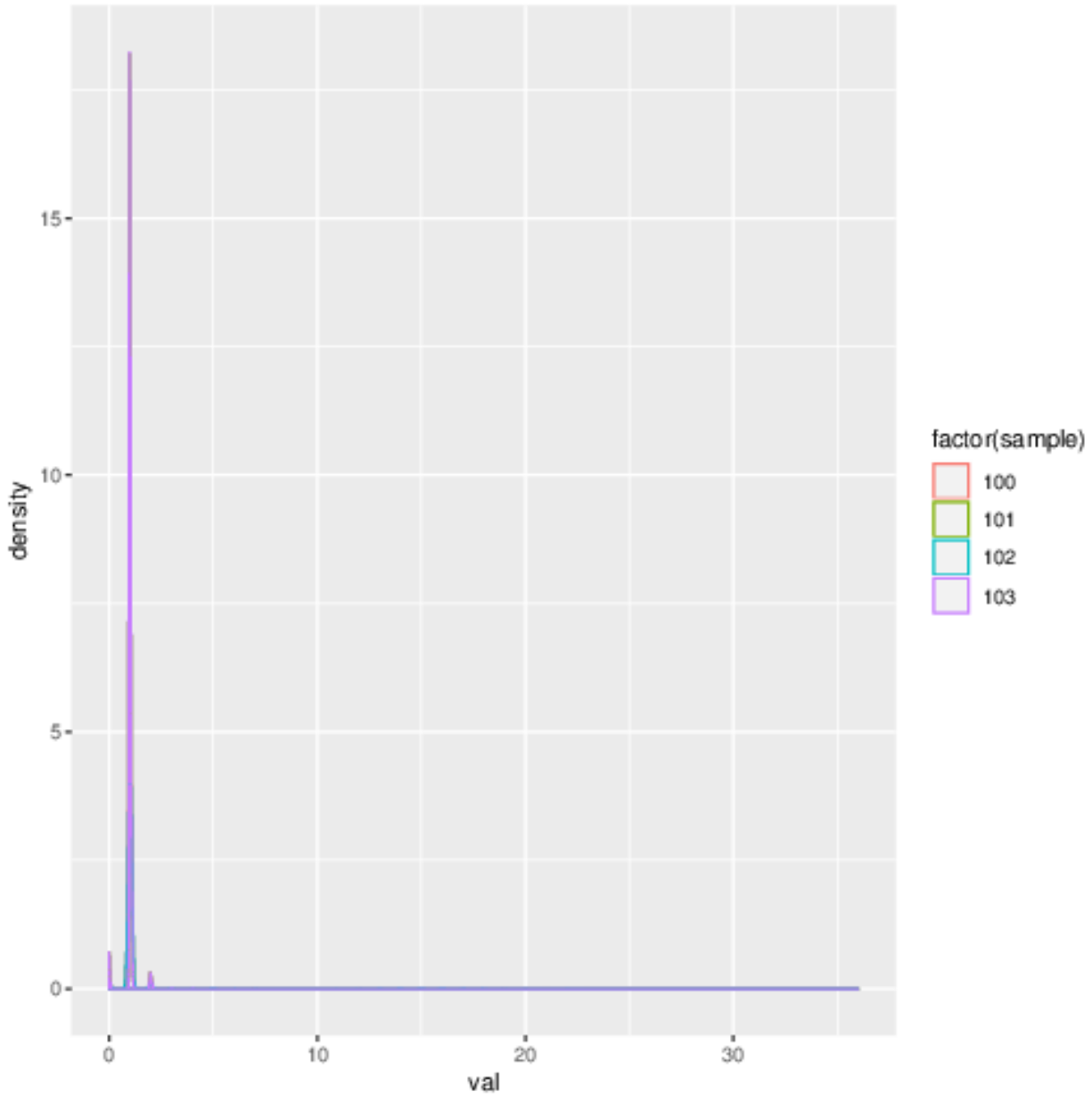


In the first stage each of the 8 input FASTQ file of the form `{sample}_{group}.fq`, where sample can be any of 100,101,102,103 and group can be 1 or 2, is aligned to a reference genome and creates a temporary alignment file.

In the second stage, for each sample the two temporary alignment files for the two groups are converted to a BAM alignment file.

In the third stage, a genome coverage table is generated for each sample.

In the final stage, the coverage tables for the four samples are consolidated into a single coverage table. A a plot containing the coverage densities of the four samples as shown below is also created.


density
15
10
5
0
0
10
20
30
val
factor(sample)
100
101
102
103

CHAPTER 7

JUDI Code

Here we demonstrate step by step how to build the pipeline described here using JUDI.

7.1 Parameter database

The pipeline has two parameters: `sample` with possible values 100,101,102,103 and `group` with possible values 1,2. The global parameter database for the pipeline can be created as follows:

```
add_param('100 101 102 103'.split(), 'sample')
add_param('1 2'.split(), 'group')
```

7.2 Files

The pipeline deals with 5 types of files which can defined through the following 5 JUDI files.

- reads: 8 input FASTQ files, each corresponding to a row of the global parameter database.

```
path_gen = lambda x: '{}_{}.fq'.format(x['sample'],x['group'])
reads = File('orig_fastq', path = path_gen)
```

- sai: 8 temporary alignment files that are created in the first stage of the pipeline. Its parameter database is same as the global parameter database.

```
sai = File('aln.sai')
```

- bam: 4 alignment files, one for each sample, that are created in the second stage of the pipeline. Its parameter database does not have parameter `group` and hence can be created by masking `group` in the global parameter database.

```
bam = File('aln.bam', mask=['group'])
```

- cov: 4 genome coverage table file in CSV format, one for each sample, that are created in the third stage of the pipeline. Its parameter database does not have parameter `group` and hence can be created by masking `group` in the global parameter database.

```
cov = File('cov.csv', mask=['group'])
```

- combined-coverage: single consolidated coverage table file in CSV format, that is created in the final stage of the pipeline. Its parameter database can be created by masking both `sample` and `group` in the global parameter database.

```
combined = File('combined.csv', mask=['sample', 'group'])
```

- plot: consolidated coverage table plotted in the final stage of the pipeline. Here, too parameter database can be created by masking both `sample` and `group` in the global parameter database. Optionally, we can store this in the current directory, instead of the default directory `judi_files`.

```
plot = File('pltcov.csv', mask=['sample', 'group'], root='.')
```

Additionally, a reference genome file used by the pipeline. However, as this file is independent of the parameters, we can keep it as a constant.

```
REF = 'hg_refs/hg19.fa'
```

7.3 Tasks

Each of the four stages of the pipeline is implemented as a JUDI task.

- Align FASTQ: We need to align each FASTQ separately. So we create a task with parameter database same as the global parameter database.

```
class AlignFastq(Task):
  inputs = {'reads': File('orig_fastq', path = path_gen)}
  targets = {'sai': File('aln.sai')}
  actions = [('bwa aln {} {} > {}', [REF,'$reads','$sai'])]
```

- Create BAM: We need one task instance for each sample. So we create a task with only `sample` as the parameter, or alternatively by masking `group` from the global parameter database. We reuse the files defined in the `AlignFastq` task.

```
class CreateBam(Task):
  mask = ['group']
  inputs = {'reads': AlignFastq.inputs['reads'],
            'sai': AlignFastq.targets['sai']}
  targets = {'bam': File('aln.bam', mask = mask)}
  actions = [('bwa sampe {} {} {} | samtools view -Sbh - | samtools sort - > {}',␣
↪[REF,'$sai','$reads','$bam'])]
```

- Get coverage: We need one task instance for each sample. So we create a task with `group` masked from the global parameter database.

```
class GetCoverage(Task):
  mask = ['group']
  inputs = {'bam': CreateBam.targets['bam']}
  targets = {'cov': File('cov.csv', mask = mask)}
  actions = [('(echo val; samtools rmdup {} - | samtools mpileup - | cut -f4) > {}', [
↪'$bam','$cov'])]
```

(continues on next page)

(continued from previous page)

- Combine Coverage: We need only task instance. So we create a task with both `sample` and `group` masked from the global parameter database.

```
class CombineCoverage(Task):
  mask = ['group', 'sample']
  inputs = {'cov': GetCoverage.targets['cov']}
  targets = {'csv': File('combined.csv', mask = mask),
             'pdf': File('pltcov.pdf', mask = mask, root = '.')}
  actions = [(combine_csvs, ['#cov', '#csv']),
             ("""echo "library(ggplot2); pdf('{}')
              ggplot(read.csv('{}'), aes(x = val)) +
              geom_density(aes(color = factor(sample)))"\
              | R --vanilla""", ['$pdf','$csv'])]
```

CHAPTER 8

Execution

The pipeline built here could be put all together in a `dodo.py` file:

```
from judi import File, Task, add_param, combine_csvs

add_param('100 101 102 103'.split(), 'sample')
add_param('1 2'.split(), 'group')

REF = 'hg_refs/hg19.fa'
path_gen = lambda x: '{}_{}.fq'.format(x['sample'],x['group'])

class AlignFastq(Task):
  inputs = {'reads': File('orig_fastq', path = path_gen)}
  targets = {'sai': File('aln.sai')}
  actions = [('bwa aln {} {} > {}', [REF,'$reads','$sai'])]

class CreateBam(Task):
  mask = ['group']
  inputs = {'reads': AlignFastq.inputs['reads'],
            'sai': AlignFastq.targets['sai']}
  targets = {'bam': File('aln.bam', mask = mask)}
  actions = [('bwa sampe {} {} {} | samtools view -Sbh - | samtools sort - > {}',␣
↪[REF,'$sai','$reads','$bam'])]

class GetCoverage(Task):
  mask = ['group']
  inputs = {'bam': CreateBam.targets['bam']}
  targets = {'cov': File('cov.csv', mask = mask)}
  actions = [('(echo val; samtools rmdup {} - | samtools mpileup - | cut -f4) > {}', [
↪'$bam','$cov'])]

class CombineCoverage(Task):
  mask = ['group', 'sample']
  inputs = {'cov': GetCoverage.targets['cov']}
  targets = {'csv': File('combined.csv', mask = mask),
```

(continues on next page)

(continued from previous page)

```
          'pdf': File('pltcov.pdf', mask = mask, root = '.')}
  actions = [(combine_csvs, ['#cov', '#csv']),
             ("""echo "library(ggplot2); pdf('{}')
              ggplot(read.csv('{}'), aes(x = val)) +
              geom_density(aes(color = factor(sample)))"\
              | R --vanilla""", ['$pdf','$csv'])]
```

And then executed as follows:

```
$ doit -f dodo.py
```

The pipeline can be run using more than one processor by using `-n 8` command line option to `doit`.

CHAPTER 9

List Pipeline Stages

The `list` command in DoIt can be used to query the tasks created in a `dodo` script.

For example, `doit list -f dodo.py` on the dodo script described here gives the following output.

```
$ doit list -f dodo.py
CombineCounts
GetCounts
Task
```

Note that `Task` here is the default task that all other tasks inherit from. It is also possible to list and see the status of the task instances, whether to rebuild etc., using the `--all --status` options of `doit list` command.

```
$ doit list --all --status -f dodo.py
R CombineCounts
U CombineCounts:   Combine counts
R GetCounts
U GetCounts:n~1    Count lines, words and characters in file
U GetCounts:n~2    Count lines, words and characters in file
R Task
```

where `R` denotes 'to run' and `U` denotes 'to update'. More details on the `doit list` command can be found here.

CHAPTER 10

Cleanup Files

It is possible to selectively delete the physical files created by a JUDI task or a task instance. For example, `doit clean -f dodo.py CombineCounts` on the dodo script created in this example deletes the result file from the final stage of the pipeline.

```
$ doit clean -f dodo.py CombineCounts
CombineCounts: - removing file './result.csv'
```

The following example shows how to clean the files created by a single task instance.

```
$ doit clean -f dodo.py GetCounts:n~1
GetCounts:n~1 - removing file './judi_files/n~1/counts.csv'
```

More details about DoIt clean command can be found here.

CHAPTER 11

More DoIt Features

JUDI is implemented using the class example given at the blog maintained by the creator of DoIt.

JUDI changes only the way the DoIt tasks are created, the rest of the features of DoIt such as execution etc. are kept unchanged in JUDI. We would like to encourage you to explore other features of DoIt described here.

Some recommended features are:

- verbosity
- task info
- uptodate
- etc.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index

Symbols

A

F

J

S