University of Zurich[UZH]

S3IT

Running tasks in sequence: `SequentialTaskCollection` and `StagedTaskCollection`

Riccardo Murri <riccardo.murri@uzh.ch>

S3IT: Services and Support for Science IT

University of Zurich

Aug. 28 – Sep. 1, 2017

Basic use of SequentialTaskCollection

```
from gc3libs.workflow \
  import SequentialTaskCollection

class MySequence (SequentialTaskCollection) :
  # ...
  def __init__(self, ...):
    app1 = FirstApp(...)
    app2 = SecondApp(...)
    SequentialTaskCollection.__init__(
      self, [app1, app2])
```

A `SequentialTaskCollection` runs a list of tasks one at a time, in the order given.

Basic use of SequentialTaskCollection

```
from gc3libs.workflow \
  import SequentialTaskCollection

class MySequence(SequentialTaskCollection):
  # ...
  def __init__(self, ...):
    app1 = FirstApp(...)
    app2 = SecondApp(...)
    SequentialTaskCollection.__init__(
      self, [app1, app2])
```

Initialize a `SequentialTaskCollection` with a list of tasks to run.

Running tasks in sequence

```
class MyScript(SessionBasedScript):
  # ...
  def new_tasks(self, extra):
    tasks_to_run = [
      MySequence(...)
    ]
    return tasks_to_run
```

You can then run the entire sequence by returning it from `new_tasks()`.

Exercise 8.A:

Write a `priceplot.py` script that performs the following two steps:

1. Run the `simAsset.R` script (from Exercise 6.D) with the parameters given on the command line, and
2. Feed the `results.csv` file it outputs into the `saplot.py` script and retrieve the produced `saplot.pdf` file.

Run it like `simAsset.R`, for example:

```
$ python priceplot.py 50 0.04 0.1 0.27 10 40
```

Running jobs in sequence

`StagedTaskCollection` provides a simple interface for constructing sequences of tasks, but only when the number and content of steps is *known and fixed* at programming time.

(By contrast, the most general `SequentialTaskCollection` can alter the sequence on the fly, insert new stages while running and loop back. But the code is also harder to write.)

```
class Pipeline (StagedTaskCollection) :
  def __init__(self, image):
    self.source = image
    # super() must come *last*
    super(Pipeline).__init__(self)

  def stage0(self):
    # ...

  def stage1(self):
    # ...

  # ...
  def stageX(self):
    # ...
```

Example: subclassing a `StagedTaskCollection`

```
class Pipeline(StagedTaskCollection):
  def __init__(self, image):
    self.source = image
    # super() must come *last*
    super(Pipeline).__init__(self)

  def stage0(self):
    # ...

  def stage1(self):
    # ...

  # ...
  def stageX(self):
    # ...
```

Stages are numbered starting from 0.

You can have as many stages as you want.

```
class Pipeline(StagedTaskCollection):
  # ...

  def stage0(self):
    # run 1st step
    return Application(
      ['convert', self.source,
       '-colorspace', 'gray',
       'grayscale_' + self.source],
      inputs = [self.source],
      ...)

  # ...
```

Each `stageX` method can return a `Task` instance, that will run as the X-th step in the sequence.

```
class Pipeline(StagedTaskCollection):
  # ...

  def stage1(self):
    if self.tasks[0].execution.exitcode != 0:
      # bail out
      return (0, 1)
    else:
      # run 2nd step
      return Application(...)

  # ...
  def stageX(self):
    # ...
```

In later stages you can check the exit code of earlier ones, and decide whether to continue the sequence or abort.

```
class Pipeline(StagedTaskCollection):
  # ...

  def stage1(self):
    if ...:
      # bail out
      return (0, 1)
    else:
      # run 2nd step
      return Application(...)

  # ...
  def stageX(self):
    # ...
```

To abort the sequence, return an integer (termination status) or a pair *(signal, exit code)*, instead of a `Task` instance.

This sets the collections' own signal and exit code, and also sets the state as `TERMINATED`.

Detour: BLAST, again

Another use of the BLAST tool is to search for given "query" proteins in a data base. Large curated DBs are available, but one may want to build a custom DB.

Building a DB from a set of FASTA-format files `p1.faa` `p2.faa` and `p3.faa`, and querying it is a 3-step process:

```
cat p1.faa p2.faa p3.faa > db.faa
formatdb -i db.faa
blastpgp -i q.faa -d db.faa -e ...
```

The `formatdb` step produces output files `db.faa.phr`, `db.faa.pin`, and `db.faa.psq`; all these files are *inputs* to the `blastpgp` program.

Exercise 8.B: *(Difficult)*
Write a `blastdb.py` script to build a BLAST DB and query it.

The `blastdb.py` script shall be invoked like this:

```
$ python blastdb.py query.faa p1.faa [p2.faa ...]
```

where arguments `new.faa`, `p1.faa`, etc. are FASTA-format files.

The script should build a BLAST DB out of the files p*N*.faa. Then, it should query this database for occurrences of the proteins in `query.faa` using `blastpgp`.

Exercise 8.C: Find out by running the `blastdb.py` script of Ex. 8.B:

1. What happens if an intermediate step fails and does not produce complete output?

2. After the whole sequence turns to TERMINATED state, what is the value of its signal and exitcode?

Exercise 8.D: Implement (in `blastdb.py`) a "cleanup" feature that removes intermediate results (e.g., the "`.phr`" files) and only keeps the output from `blastpgp` *if the whole sequence was successfully executed*.