Batch Q

Release 0.1

Troels F. Rønnow

Nov 08, 2017

Contents

CHAPTER 1

Introduction

BatchQ is a set of classes written in Python which aim toward automating all kinds of task. BatchQ was designed for interacting with terminal application including Bash, SSH, SFTP, Maple, Mathematica, Octave, Python and more. If for one or another reason your favorite application is not supported BatchQ is easily extended to support more programs.

1.1 Dependencies and compatibility

Currently BatchQ only depends on the libraries shipped with Python. It should therefore work out of the box.

Please note that at the present moment BatchQ has only been tested with Python 2.7 on Mac OS X 10.6 and with Python 2.6.5 on Ubuntu 10.10.

1.2 Installation

There are currently two ways of installing BatchQ. Either you use setup tools in which case you write

```
easy_install batchq
```

Alternatively you can download the latest version from GitHub (.zip,‘.tar.gz‘_) and install using setup.py

```
cd [location/of/source]
python setup.py install [ --user | --home=~ ]
```

The `--user` and `--home` flags are optional and are intended for users who do not have write access to the global system. More information can be found the Python install page.

To test your installation type:

```
q help
```

If a help message is displayed you are ready to go on and submit your first job.

1.3 Manual installation from GitHub

The manual installation is intended for development purposes and for persons who do not want to rely on the install script:

```
export INSTALL_DIR=~/Documents       # Change if you want another install location
cd $INSTALL_DIR
git clone https://github.com/troelsfr/BatchQ.git
echo export PYTHONPATH=\$PYTHONPATH:$INSTALL_DIR/BatchQ/ >> ~/.profile
echo export PATH=\$PATH:$INSTALL_DIR/BatchQ/batchq/bin >> ~/.profile
```

Note that `.profile` may be named differently on your system, i.e. `.bashrc` or `.profilerc`. Start a new session of bash and write

```
q list
```

If a list of available commands is displayed, you have successfully installed BatchQ.

1.4 Note for Windows users

This version of BatchQ is not yet supported on Windows platforms. Developers are encouraged to extend the `Process` module. Unfortunately, it seems that it is not possible to create a pure Python solution and a terminal module should be written in C/C++.

For information on the structure of the code please consult the developers introduction.

CHAPTER 2

Users Guide

In the following we will refer to a tool called `nohup`. This name is missleading as we do not use the actual nohup tool for several reasons, the most significant being that nohup some times hangs up. Instead we start a background job in a subshell, i.e.

```
$ (./job_name parameters & )
```

This essentially have the desired effect, namely that the program does not hang up when the shell is closed. Throughout this document we will refer to this as `nohup` rather than `starting a background job in a subshell`. For more information on nohup alternatives see What to do when nohup hangs up anyway.

2.1 Quick start

This section is intended for getting you started quickly with BatchQ and consequently, few or no explanations of the commands/scripts will be given. If you would like the full explanation on how BatchQ works, skip this section. If you choose to read the quick start read all of it no matter whether you prefer Python over bash.

2.1.1 Command line

First you need to create configurations for the machines you want to access. This is not necessary, but convenient (more details are given in the following sections). Open `bash` and type

```
$ q configuration my_server_configuration  --working_directory="Submission" --command=
↪"./script"  --input_directory="." --port=22 --server="server.address.com" --global
$ q configuration your_name  --username="your_default_used"  --global
```

In the above change the `server.address.com` to the server address you wish to access. Also, change the `username` in the second line to your default username. Next, create a new director `MyFirstSubmission` and download the script `sleepy`

```
mkdir MyFirstSubmission
cd MyFirstSubmission
wget https://raw.github.com/troelsfr/BatchQ/master/scripts/sleepy
chmod +x sleepy
```

The job `sleepy` sleeps for 100 seconds every and for every second it echos "Hello world". Submit it using `server.address.com` using the command:

```
$ q [batch_system] job@my_server_configuration,your_name --command="./sleepy"
```

Here `batch_system` should be either `nohup`, `ssh-nohup` or `lsf`. Check the status of the job with

```
$ q [batch_system] job@my_server_configuration,your_name --command="./sleepy"
Job is running.
```

And after 100s you get

```
$ q [batch_system] job@my_server_configuration,your_name --command="./sleepy"
Job has finished.
Retrieving results.
```

At this point new files should appear in your current directory:

```
$ ls
sleepy   sleepy.data
```

In order to see the logs of the submission type

```
$ q [batch_system] stdout@my_server_configuration,your_name --command="./sleepy"
This is the sleepy stdout.
$ q [batch_system] stderr@my_server_configuration,your_name --command="./sleepy"
This is the sleepy stderr.
$ q [batch_system] log@my_server_configuration,your_name --command="./sleepy"
(...)
```

The last command will differ depending on which submission system you use. Finally, we clean up on the server:

```
$ q [batch_system] delete@my_server_configuration,your_name --command="./sleepy"
True
```

Congratulations! You have submitted your first job using the command line tool.

2.1.2 Python

Next, open an editor, enter the following Python code:

```
from batchq.queues import  LSFBSub
from batchq.core.batch import DescriptorQ

class ServerDescriptor(DescriptorQ):
  queue = LSFBSub
  username = "default_user"
  server="server.address.com"
  port=22
  prior = "module load open_mpi goto2 python hdf5 cmake mkl\nexport PATH=$PATH:$HOME/
↪opt/alps/bin"
```

```
  working_directory = "Submission"

desc1 = ServerDescriptor(username="tronnow",command="./sleepy 1", input_directory=".",
↪ output_directory=".")
desc2 = ServerDescriptor(desc1, command="./sleepy 2", input_directory=".", output_
↪directory=".")

print "Handling job 1"
desc1.job()
print "Handling job 2"
desc2.job()
```

and save it as `job_submitter.py` in `MyFirstSubmission`. Note that in the above code we use the second descriptor to initiate the first descriptor in order to reuse the queue defined for `desc1` in `desc2`. Go back to the shell and type:

```
$ python job_submitter.py
```

Rerun the code to get the status of the job and to pull finished jobs. Your second submission was done with Python.

Note: If choosing same input and output directory you will run into problems when running this script several times as hash sum of the input changes once the results have been pulled. This means that you may accidently resubmit a finished job.

The above can be overcome by either separating `input_directory` and `output_directory`, or by setting the submission id manually:

```
from batchq.queues import  LSFBSub
from batchq.core.batch import Descriptor as DescriptorQ

class ServerDescriptor(DescriptorQ):
  queue = LSFBSub
  username = "default_user"
  server="server.address.com"
  port=22
  prior = "module load open_mpi goto2 python hdf5 cmake mkl\nexport PATH=$PATH:$HOME/
↪opt/alps/bin"
  working_directory = "Submission"

desc1 = ServerDescriptor(username="tronnow",command="./sleepy 1", input_directory=".",
↪ output_directory=".", overwrite_submission_id="simu1")
desc2 = ServerDescriptor(desc1, command="./sleepy 2", input_directory=".", output_
↪directory=".", overwrite_submission_id="simu2")

print "Handling job 1"
desc1.job()
print "Handling job 2"
desc2.job()
```

To shorten the above you may use your previously defined configurations

```
from batchq.queues import  LSFBSub
from batchq.core.batch import DescriptorQ, load_queue

q = load_queue(LSFBSub, "my_server_configuration,your_name")
```

```
desc1 = DescriptorQ(q, command="./sleepy 1", input_directory=".", output_directory=".
↪", overwrite_submission_id="simu1")
desc2 = DescriptorQ(q, command="./sleepy 2", input_directory=".", output_directory=".
↪", overwrite_submission_id="simu2")

print "Handling job 1"
desc1.job()
print "Handling job 2"
desc2.job()
```

We can now generalise this to arbitrarily many jobs:

```
from batchq.queues import  LSFBSub
from batchq.core.batch import DescriptorQ, load_queue

q = load_queue(LSFBSub, "my_server_configuration,your_name")
for i in range(1,10):
    desc = DescriptorQ(q, command="./sleepy %d" %i, input_directory=".", output_
↪directory=".", overwrite_submission_id="simu%d" %i)

    print "Handling job %d" %i
    desc.job()
```

If we know that the output files does not overwrite each other it is only necessary to keep one copy of the input folder. This can be done by specifying the `subdirectory`

```
from batchq.queues import  LSFBSub
from batchq.core.batch import DescriptorQ, load_queue

q = load_queue(LSFBSub, "my_server_configuration,your_name")
for i in range(1,15):
    desc = DescriptorQ(q, command="./sleepy %d" %i, input_directory=".", output_
↪directory=".", overwrite_submission_id="simu%d" %i, subdirectory="mysimulation")
    print "Handling job %d" %i
    desc.job()
```

2.2 Using the command line tool

The following section will treat usage of BatchQ from the command line.

2.2.1 Available modules

The modules available to BatchQ will vary from system to system depending on whether custom modules have been installed. Modules are divided into four categories: functions, queues, pipelines and templates. The general syntax of the Q command is:

```
$ q [function/queue/template] [arguments]
```

The following functions are available through the console interface and using Python and are standard modules included in BatchQ which provides information about other modules

2.2.2 Submitting jobs

The BatchQ command line interface provides you with two predefined submission modules: `nohup` and `lsf`. `nohup` is available on every

To submit a job type:

```
$ cd /path/to/input/directory
$ q lsf submit -i --username=user --working_directory="Submission" --command="./script
↪" --input_directory="." --port=22 --server="server.address.com"
```

The above command will attempt to log on to `server.address.com` using the username `user` through port 22. It then creates a working directory called `Submission` in the entrance folder (usually your home directory on the server) and transfer all the files from your `input_directory` to this folder. The `command` is then submitted to `lsf` and the SSH connection is terminated.

Once you have automated the submission process you want to store the configuration parameters in a file in order to shorten the commands need to operate on your submissions. Using the example from before, this can be done as

```
$ q configuration brutus -i --username=user --working_directory="Submission" --
↪command="./script" --input_directory="." --port=22 --server="server.address.com"
```

The above code creates a configuration named "brutus" which contains the instructions for submitting your job on "server.address.com". Having created a configuration file you can now submit jobs and check status with

```
$ q lsf submit@brutus
True
$ q lsf pid@brutus
12452
```

This keeps things short and simple. You will need to create a configuration file for each server you want to submit your job. If for one or another reason you temporarily want to change parameters of your configuration, say the `working_directory`, this can be done by adding a long parameter:

```
$ q lsf submit@brutus --working_directory="Submission2"
True
```

You can configure Batch Q command line tool with several input configurations

Checking the status of a job, retrieving results and deleting the working directory of a simulation is now equally simple

```
$ q lsf status@brutus
DONE

$ q lsf recv@brutus
True

$ q lsf delete@brutus
True
```

The retrieve command will only retrieves files that does not exist, or differs from those in the input directory.

Finally, the Q system implements a fully automated job submission meaning that the system will try to determine the state of you job and take action accordingly. For fast job submission and status checking write:

```
$ q lsf job@brutus,config
Uploading input directory.
Submitted job on brutus.ethz.ch
```

```
$ q lsf job@brutus,config
Job pending on brutus.ethz.ch

$ q lsf job@brutus,config
Job running on brutus.ethz.ch

$ q lsf job@brutus,config
Job finished on brutus.ethz.ch
Retrieving results.

Do you want to remove the directory 'Submission2' on brutus.ethz.ch (Y/N)? Y
Deleted Submission2 on brutus.ethz.ch
```

You can equally submit the job on your local machine using `nohup` instead of `lsf`.

2.2.3 A few words on job hashing

When submitting a job Batch Q generates a hash for your job. The hash includes following:

- An MD5/SHA sum of the input directory
- The name of the server to which the job is submitted
- The submitted command (including parameters)

It is not recommended, nevertheless possible, to overwrite the hash key. This can be done by adding a `--overwrite_submission_id="your_custom_id"`. This can be useful in some cases. For instance you might want to work on your source code during development. This would consequently changed the MD5 of your input directory and Batch Q would be incapable of recognising your job submission. Batch Q is shipped with a configuration for debugging which can be invoked by

```
$ q lsf submit@brutus,debug
```

The debug configuration is only suitable for debug as the submission id is `debug`.

Another scenario where you may want to change the hashing routine is the case where you store your output data in your input directory. Submitting several jobs and pulling results will over time change the hash of the input directory. To overcome this issue add `eio` (short for equal input/output directory) to your configuration

```
$ q lsf submit@brutus,eio
```

The `eio` flag will overwrite your `output_directory` with the value of your `input_directory` and change the hashing routine to only include the command and server name.

2.2.4 Example: Submitting ALPS jobs using nohup

2.2.5 Example: Submitting ALPS jobs using LSF

2.2.6 Example: Submitting multiple jobs from one submission directory

In some cases one may not want to copy the same directory several times to the server as this may take up vast amounts of space. If the simulation output only depends the command line parameters (as is the case for ALPS `spinmc`) one can use the `eio` configuration to submit several commands reusing the same submission directory

```
$ q lsf job@brutus,eio --working_directory="Submission" --command="spinmc TODO1"
$ q lsf job@brutus,eio --working_directory="Submission" --command="spinmc TODO2"
$ q lsf job@brutus,eio --working_directory="Submission" --command="spinmc TODO3"
```

2.3 Using Python

2.3.1 Submitting jobs

2.3.2 Retrieving results

2.3.3 Q descriptors, Q holders and Q functions

Batch Q user API is based on three main classes Q descriptors, Q holders (queues) and Q functions. Usually Q functions are members of instances of Q holder classes while Q descriptors are reference objects used to ensure that you do not open more SSH connections than necessary. Descriptors link a set of input configuration parameters to a given queue. An example could be:

```
class ServerDescriptor(DescriptorQ):
      queue = LSFBSub
      username = "user"
      server="server.address.com"
      port=22
      options = ""
      prior = "module load open_mpi goto2 python hdf5 cmake mkl\nexport PATH=$PATH:
↪$HOME/opt/alps/bin"
      post = ""
      working_directory = "Submission"
```

The descriptor `ServerDescriptor` implements all Q functions and properties defined in the class `LSFBSub`. However, the descriptor ensures that the all queue parameters are set accordingly to those given by the descriptor definition before executing a command on the queue. Therefore, if you have two descriptor instances that shares a queue

```
queue = LSFBSub()
desc1 = DescriptorQ(queue)
desc1.update_configuration(working_directory = "Submission1")
desc2 = DescriptorQ(queue)
desc2.update_configuration(working_directory = "Submission2")
```

you are ensured that your are working in the correct directory by using the descriptor instead of the queue directly. Notice that in order to update the queue properties using the descriptor one needs to use `update_configuration` rather than `descriptor.property` (i.e. `desc2.working_directory` in the above example). The reason for this is that any method or property of a descriptor object is an "reference" to queue methods and properties. The only methods that are not redirected are the implemented descriptor methods:

```
desc1 = ServerDescriptor()
desc2 = ServerDescriptor(desc2)
desc2.update_configuration(working_directory = "Submission2")
```

In general, when copying descriptors, make sure to do a shallow copy as you do not want to make a deep copy of the queue object.

2.3.4 Example: Submitting ALPS jobs using nohup

The BatchQ package comes with a preprogrammed package for ALPS. This enables easy and fast scripting for submitting background jobs on local and remote machines. Our starting points is the Spin MC example from the ALPS documentation:

```
import pyalps
import matplotlib.pyplot as plt
import pyalps.plot
import sys

print "Starting"

parms = []
for t in [1.5,2,2.5]:
   parms.append(
       {
         'LATTICE'        : "square lattice",
         'T'              : t,
         'J'              : 1 ,
         'THERMALIZATION' : 1000,
         'SWEEPS'         : 100000,
         'UPDATE'         : "cluster",
         'MODEL'          : "Ising",
         'L'              : 8
       }
   )

input_file = pyalps.writeInputFiles('parm1',parms)
desc = pyalps.runApplication('spinmc',input_file,Tmin=5,writexml=True)

result_files = pyalps.getResultFiles(prefix='parm1')
print result_files
print pyalps.loadObservableList(result_files)
data = pyalps.loadMeasurements(result_files,['|Magnetization|','Magnetization^2'])
print data
plotdata = pyalps.collectXY(data,'T','|Magnetization|')
plt.figure()
pyalps.plot.plot(plotdata)
plt.xlim(0,3)
plt.ylim(0,1)
plt.title('Ising model')
plt.show()
print pyalps.plot.convertToText(plotdata)
print pyalps.plot.makeGracePlot(plotdata)
print pyalps.plot.makeGnuplotPlot(plotdata)
binder = pyalps.DataSet()
binder.props = pyalps.dict_intersect([d[0].props for d in data])
binder.x = [d[0].props['T'] for d in data]
binder.y = [d[1].y[0]/(d[0].y[0]*d[0].y[0]) for d in data]
print binder
plt.figure()
pyalps.plot.plot(binder)
plt.xlabel('T')
plt.ylabel('Binder cumulant')
plt.show()
```

Introducing a few small changes the script now runs using BatchQ for submission:

```
from batchq.contrib.alps import runApplicationBackground, LSFBSub, DescriptorQ
import pyalps
import matplotlib.pyplot as plt
import pyalps.plot
import sys

parms = []
for t in [1.5,2,2.5]:
   parms.append(
       {
         'LATTICE'        : "square lattice",
         'T'              : t,
         'J'              : 1 ,
         'THERMALIZATION' : 1000,
         'SWEEPS'         : 100000,
         'UPDATE'         : "cluster",
         'MODEL'          : "Ising",
         'L'              : 8
       }
   )

input_file = pyalps.writeInputFiles('parm1',parms)

class Brutus(DescriptorQ):
  queue = LSFBSub
  username = "tronnow"
  server="brutus.ethz.ch"
  port=22
  options = ""
  prior = "module load open_mpi goto2 python hdf5 cmake mkl\nexport PATH=$PATH:$HOME/
↪opt/alps/bin"
  post = ""
  working_directory = "Submission"

desc = runApplicationBackground('spinmc',input_file,Tmin=5,writexml=True, descriptor␣
↪= Brutus(), force_resubmit = False )


if not desc.finished():
   print "Your simulations has not yet ended, please run this command again later."
else:
    if desc.failed():
        print "Your submission has failed"
        sys.exit(-1)
    result_files = pyalps.getResultFiles(prefix='parm1')
    print result_files
    print pyalps.loadObservableList(result_files)
    data = pyalps.loadMeasurements(result_files,['|Magnetization|','Magnetization^2'])
    print data
    plotdata = pyalps.collectXY(data,'T','|Magnetization|')
    plt.figure()
    pyalps.plot.plot(plotdata)
    plt.xlim(0,3)
    plt.ylim(0,1)
    plt.title('Ising model')
    plt.show()
    print pyalps.plot.convertToText(plotdata)
    print pyalps.plot.makeGracePlot(plotdata)
```

```
    print pyalps.plot.makeGnuplotPlot(plotdata)
    binder = pyalps.DataSet()
    binder.props = pyalps.dict_intersect([d[0].props for d in data])
    binder.x = [d[0].props['T'] for d in data]
    binder.y = [d[1].y[0]/(d[0].y[0]*d[0].y[0]) for d in data]
    print binder
    plt.figure()
    pyalps.plot.plot(binder)
    plt.xlabel('T')
    plt.ylabel('Binder cumulant')
    plt.show()
```

Executing this code submit the program as a background process on the local machine. It can easily be extended to supporting SSH and LFS by changing the queue object:

```
# TODO: Give example
```

2.3.5 Example: Submitting multiple jobs from one submission directory

2.4 Using VisTrails

2.4.1 Submitting jobs

2.4.2 Retrieving results

2.4.3 Example: Submitting ALPS jobs using nohup

2.4.4 Example: Submitting ALPS jobs using LSF

CHAPTER 3

Remote Jobs using Django

The following briefly explains how to put up a local Django webinterface and how to synchronise it with jobs submitted to a server. No particular knowledge of Django is required, except if you wish to setup a permanent web server for keeping track of jobs states.

3.1 Dependencies and preliminary notes

The web interface depends on Django 1.3 (or newer) which can be obtained using easy_install as follows

```
easy_install django
```

The web interface can be located in any directory. We will in the following assume that it is located in ~/remoteweb/.

3.2 Starting a local server

Open bash and execute following commands:

```
cd ~/remoteweb/
python manage.py syncdb
python manage.py runserver
```

This will start the local web server which can be accessed on the address http://localhost:8000/. In order to synchronise the jobs from a given server simply type

```
python mange.py update_database [appName [server[:port]]]
```

where [appName] is 'remoteJobs-vistrails' if you are using VisTrails with the standard settings and [server] is the name of the server. If no server is supplied the script will assume that you wish to work on the local machine. If a server name is supplied the script will prompt for username and password. After entering these the database is updated with the jobs which has previously been submitted to the server.

3.3 Webinterface

In order to access the web interface go to http://localhost:8000/ in your favorite browser. A list of should now appear

which has been optimised for running on mobile units. From the webinterface you can see the status of your jobs, cancel or resubmit them. The changes take effect whenever

```
python mange.py update_database [appName [server[:port]]]
```

has been executed.

CHAPTER 4

Kivy Terminal

CHAPTER 5

For Developers

5.1 Developer: Getting started

The following section is meant as an instructive example of how BatchQ can be used to automate simple tasks such as logging on to a server and creating a configuration file. This first section is somewhat basic stuff, but important, nevertheless, since BatchQ is designed to be different from your normal Python programs in its structure. All the key classes are shortly described here.

Imagine that you want to open Bash, go to Documents in your home folder and create a text file. With Bash commands for this would be

```
cd ~/Documents
echo Hello world > hello.txt
```

In this tutorial we will automate this task. If you are eager to use BatchQ and do not really care about the underlying functionality you probably should skip right away to one of the examples of job automation:

1. *Example: CreateFile*
2. *Example: CreateFileAndDirectory*
3. *Example: CreateFileSSH*

However, it is strongly recommended that you go through all the examples to get a basic understanding of the structure of BacthQ, how it works and what you can do with it.

5.1.1 Processes and Pipes

While it is not necessary to understand the concepts of Processes and Pipes in order to use BatchQ, it surely is useful in order to get a picture of the whole structure of this package. If you are familiar with *Pexpect <http://www.noah.org/wiki/pexpect>* you are already familiar with some of the key concepts of BatchQ. BatchQ provides a classes which have some similarities with Pexpect, yet differs in a number of ways, the most significant being that BatchQ communication with the terminal and expect functionality has been seperated into two classes: Process

and BasePipe. The Process class provides methods for communicating directly with a process with no intepretation of the output, whereas BasePipe implements output interpretation and expect functionality.

Unlike Pexpect, Process does not apply which on the command passed as the constructor argument and this command must be applied manually. In the following example we open a instance of Bash writes "echo Hello world" and read the output:

```
from batchq.core.process import Process
from batchq.core.utils import which

print "Starting bash found at", which("bash")
x = Process(which("bash"))

# Waits until we have the prompt
while x.getchar() != "$": pass

# Sends a command to the terminal
x.write("echo Hello world\n")

# And read the response
print ""
print "The response is:"
print x.read()
print ""
print "The full buffer is:"
print x.buffer
x.kill()
```

This code produces following output

```
Starting bash found at /bin/bash

The response is:
 echo Hello world
Hello world
bash-3.2$

The full buffer is:
[?1034hbash-3.2$ echo Hello world
Hello world
bash-3.2$
```

Here we are essentially communicating with the process at the most basic level and we have to interpret every character returned. Especially, it should be noted that escape characters are not interpreted in any way and also that the response from x.read() contains the echo of the command. Both of these features are somewhat inconvenient when ones to communicate with a process.

The BasePipe class overcome the inconvenience of the Process class by implementing expect

```
from batchq.core.process import Process
from batchq.core.communication import BasePipe
from batchq.core.utils import which

class BashTerminal(BasePipe):
    def __init__(self):
        pipe = Process(which("bash"), terminal_required = True)
        expect_token ="#-->"
        submit_token ="\n"
        initiate_pipe=True
```

```
        super(BashTerminal, self).__init__(pipe, expect_token, submit_token, initiate_
↪pipe)


    def initiate_pipe(self):
        self.pipe.write("export PS1=\"%s\"" % self._expect_token)
        self.consume_output()
        self.pipe.write(self._submit_token)
        self.consume_output()

    def echo(self, msg):
        return self.send_command("echo %s"%msg)

x = BashTerminal()
print "Response: "
print x.echo("Hello world")
print ""
print "VT100 Buffer:"
print x.buffer
print ""
print "Pipe Buffer:"
print x.pipe_buffer
```

which produces

```
Response:
Hello world


VT100 Buffer:
export PS1="#-->"bash-3.2$ export PS1="#-->"
#-->echo Hello world
Hello world
#-->

Pipe Buffer:
export PS1="#-->"[?1034hbash-3.2$ export PS1="#-->"
#-->echo Hello world
Hello world
#-->
```

The properties pipe and pipe_buffer are references to the Process object and the Process.buffer, respectively. As with the first example we see that the process buffer contains VT100 characters. However, the BasePipe uses a VT100 interpreter to ensure that the output in BasePipe.buffer looks like what you would see if you where interacting with a terminal yourself. This is important as it eases the development process of pipelines.

The separation of Process and BasePipe ensures that unsupported platforms in the future can be supported. In order to make BatchQ work on unsupported system (i.e. Windows at the moment) one has to write a new Process module for the platform. Once a Process class exists BasePipe implements all the features needed to efficiently communicate with an instance of a process.

Pipelines

Pipelines are subclasses of BasePipe which are intended for a specific program like the example in the previous section. BatchQ is shipped with many different pipelines found in batchq.pipelines which among others contain BashTerminal, SSHTerminal and SFTPTerminal. The SSHTerminal subclasses BashTerminal and replaces the process instance with

an SSH instance. This in term means that any program written for BashTerminal works with SSHTerminal which is handy as you might want to develop a program locally and once working, deploy it using SSH.

We are going to give an example of how this is done in the following code:

```
from batchq.pipelines.shell.bash import BashTerminal

class CreateFileBash(object):
    def __init__(self,directory, command):
        self.directory = directory
        self.command = command
        self.terminal = BashTerminal()

    def create_dir(self):
        home = self.terminal.home()
        self.terminal.chdir(home)

        if not self.terminal.exists(self.directory):
            self.terminal.mkdir(self.directory,True)

        self.terminal.chdir(self.directory)
        return self

    def create_file(self):
        self.create_dir()
        self.terminal.send_command(self.command)
        return self

if __name__=="__main__":
    instance = CreateFileBash("Documents/DEMO_Bash", "echo hello new directory >␣
↪hello.txt")
    instance.command = "echo Hello Python world > hello.txt"
    instance.create_file()
```

With this code we have solved the initial problem, and moreover, the code is easily extended to support SSH by subclasssing CreateFileBash and replacing the terminal:

```
from batchq.pipelines.shell.ssh import SSHTerminal

class CreateFileSSH(CreateFileBash):
    def __init__(self,directory, command, server, username, password):
        self.terminal = SSHTerminal(server, username, password)
        super(CreateFileSSH,self).__init__(directory, command)

if __name__=="__main__":
    import getpass
    user = raw_input("Username:")
    pasw = getpass.getpass()
    instance = CreateFileSSH("Documents/DEMO_SSH", "echo Hello SSH > hello.txt",
↪"localhost",user,pasw)
    instance.create_file()
```

This clearly demonstrates the power of using a standards for the various terminal implementations. The function calls used here would be also be possible to implement on a Windows machine, and thus, in this way we have a universal script that works independent of the system it is on.

5.1.2 BatchQ Basics

Next to the pipelines and processes, the BatchQ module contains five other important classes: BatchQ, WildCard, Property, Controller and Function. BatchQ is the general model which your class should inherit. It provides the functionality for manging pipelines, properties, and function queues. In short, any BatchQ script is a class that subclasses BatchQ and have one or more of WildCard, Property, Controller and Function defined. The general structure of a BatchQ class is

```
1  class SubmissionModel(batch.BatchQ):
2      param1 = Property("default value 1")
3      param2 = Property("default value 2")
4      # ...
5
6      terminal1 = Controller(Class1, arg1, arg2, ...)
7
8      t1_fnc1 = Function().a().b()
9      t1_fnc2 = Function().c().d()
10     # ...
11
12     terminal2 = Controller(Class2, arg1, arg2, ...)
13
14     t2_fnc1 = Function().e(param1).f(param2)
15     t2_fnc2 = Function().g().h()
16     # ...
17
18     def userdef1(self):
19         # ...
20         pass
21
22     def userdef2(self):
23         # ...
24         pass
```

Properties are input parameters for the automation model you are creating, controllers are pipeline holders and functions are queues of function calls that will be invoked on the class through the controller. Thus, calling `SubmissionModel().t1_fnc2()` would result in two function calls: `instance.a()` and `instance.b()` with `instance = Class1(arg1, arg2, ...)`. It is possible to use the parameters as function arguments as done in line 14. Upon realisation of the function calls `param1` and `param2` are fetched. They may be changed using the normal assignment operator before the function call to change their value. Get an idea how this works we will in the following go through some simple examples.

Example: Hello world

In this example we demonstrate how a function queue is invoked on the pipeline:

```
from batchq.core import batch
class Pipeline(object):

    def hello_world(self):
        print "Hello world"

    def hello_batchq(self):
        print "Hello BatchQ"

class HelloWorld(batch.BatchQ):
    ctrl = batch.Controller(Pipeline)
```

```
    fnc = batch.Function().hello_world().hello_batchq()

instance = HelloWorld()
instance.fnc()
```

This code results in following output

```
Hello world
Hello BatchQ
```

What happens is:

1. A controller is defined which creates an instance `pipe` of the Pipeline class.
2. Next a function `fnc` is defined as the sequence of function calls ".hello_world(), .hello_batchq()"
3. When `fnc` is called two function calls, `pipe.hello_world()` and `pipe.hello_batchq()` are made.

It is worth noting that the function queue is abstract, meaning that the queue accept any calls - errors will first occur when `instance.fnc()` is called.

Example: Hello parameter

Next we extend the previous program introducing a parameter

```
from batchq.core import batch
class Pipeline(object):

    def hello(self,who):
        print "Hello",who


class HelloParam(batch.BatchQ):
    message = batch.Property("Parameter")
    ctrl = batch.Controller(Pipeline)
    fnc = batch.Function().hello(message)

HelloParam().fnc()
ins1 = HelloParam("Second Instance")
ins2 = HelloParam()
ins2.message = "Property"
ins2.fnc()
ins1.fnc()
```

Three lines of output are produces here:

```
Hello Parameter
Hello Property
Hello Second Instance
```

each corresponding to a call to `fnc()`. This code shows how a parameter can be passed on to a function call. It is important to notice that the parameter can be changed even after creating an instance of an object and that properties are *not* static members of the class, i.e. different instances may have different values. Finally, it is worth noticing that properties becomes constructor arguments of the BatchQ subclass.

Example: Hello error I

The default of BatchQ is that all errors raised are suppressed by the Function queue. You can, however, force BatchQ to raise errors by setting the parameter `verbose` of the function object:

```
from batchq.core import batch
class ErrorTest(batch.BatchQ):
    ctrl = batch.Controller(object)
    fnc1 = batch.Function(verbose=True).hello_world("Hello error")
    fnc2 = batch.Function(verbose=False).hello_world("Hello error")

instance = ErrorTest()
try:
    instance.fnc1()
except:
    print "An error occured in fnc1"
try:
    instance.fnc2()
except:
    print "An error occured in fnc2"
```

Example: Hello error II

The BatchQ function also is given the possibility to raise errors. It is done in the following way:

```
from batchq.core import batch

class Model(batch.BatchQ):
    ctrl = batch.Controller(object)
    fnc1 = batch.Function().Qthrow(exception = StandardError("Custom error class"))
    fnc2 = batch.Function().Qthrow("Standard error")

Model().fnc1()
Model().fnc2()
```

The `throw` function ignores the `verbose` option on the `Function` object and is always thrown.

Example: Function Inheritance

Common jobs such as changing into a working directory is often done in several procedures when making a batch script. For this reason BatchQ allows you to inherit earlier defined functions

```
from batchq.core import batch

class Pipe(object):
    def display(self, msg):
        print msg

class Inherits(batch.BatchQ):
    ctrl = batch.Controller(Pipe)
    fnc1 = batch.Function().display("Hello world")
    fnc2 = batch.Function(fnc1).display("Hello world II")

Inherits().fnc2()
```

Upon inheritance the entire queue from `fnc1` is copied meaning that the result stack of `fnc1` is not populated as it would have been with a function call.

Example: Function Calls

Often it is useful to divide jobs into subprocedures. For this purpose it can often be necessary to do function calls. Function calls are also allowed with BatchQ

```
from batchq.core import batch

class Pipe(object):
    def display(self, msg):
        print msg
        return msg

class CallIt1(batch.BatchQ):
    ctrl = batch.Controller(Pipe)
    fnc1 = batch.Function().display("Hello world")
    fnc2 = batch.Function().display("Before call").Qcall(fnc1).display("After call") \
        .Qprint_stack()

CallIt1().fnc2()
print ""

class CallIt2(batch.BatchQ):
    _ = batch.WildCard()
    ctrl = batch.Controller(Pipe)
    fnc1 = batch.Function().display("Hello world")
    fnc2 = batch.Function().Qjoin(fnc1, " II").display(_)

CallIt2().fnc2()
```

One important thing to note here is that it is the Function object that is passed on as argument and not Function(). The reason is that we are creating a reference model, the queue, and not performing an actual call during the construction of the class.

Wildcards

TODO: Explain the concept of a wildcard

In the following example we define five different wild cards. The first wildcard _ pops the result stack and the second _r pops the result stack in reverse order - that is, if _, _ produces “a”, “b”, then _r, _r produces “b”,”a”.

```
from batchq.core import batch

class TestPipe4(object):
    fnc1 = lambda self: "Function 1"
    fnc2 = lambda self: "Function 2"
    fnc3 = lambda self: "Function 3"
    fnc4 = lambda self: "Function 4"
    fnc5 = lambda self: "Function 5"

    def display(self, msg):
        print "1: ", msg
        return "Display function I"
```

```
    def display2(self, msg1, msg2):
        print "2: ", msg1, msg2
        return "Display Function II"


class Q6(batch.BatchQ):
    _ = batch.WildCard()
    _r = batch.WildCard(reverse = True)
    _l = batch.WildCard(lifo = False)
    _lr = batch.WildCard(lifo = False, reverse = True)
    _3 = batch.WildCard(select = 3)

    pipe = batch.Controller(TestPipe4)

    call_fnc = batch.Function() \
        .fnc1().fnc2().fnc3() \
        .fnc4().fnc5()

    test1 = batch.Function(call_fnc).display(_)
    test2 = batch.Function(call_fnc).display2(_,_)
    test3 = batch.Function(call_fnc).display2(_r,_r)
    test4 = batch.Function(call_fnc).display2(_l,_l)
    test5 = batch.Function(call_fnc).display2(_lr,_lr)
    test6 = batch.Function(call_fnc).display(_3)


instance = Q6()
instance.test1()
instance.test2()
instance.test3()
instance.test4()
instance.test5()
instance.test6()
```

Example: CreateFile

At this point we have significantly insight into the structure of BatchQ to write a realisation of the initial problem: creating a script that creates a directory:

```
from batchq.core import batch
from batchq.pipelines.shell.bash import BashTerminal
from batchq.pipelines.shell.ssh import SSHTerminal

class CreateFile1(batch.BatchQ):
  _ = batch.WildCard()

  directory = batch.Property()
  command = batch.Property()

  terminal = batch.Pipeline(BashTerminal)

  create_file = batch.Function() \
      .home().chdir(_) \
      .chdir(directory).send_command(command)
```

```
instance = CreateFile1("Documents", "echo hello world > hello.txt")
instance.create_file()
```

While this code is short and very easy to read, it obviously have shortcomings: The code will fail if the directory does not already exist. In order to resolve this issue we need to use some built-in functions to do simple if statements.

Built-in Do-function

Example: CreateFileAndDirectory

With the `do` we can now extend our previous script and ensure that the directory is created if it does not exist:

```
from batchq.core import batch
from batchq.pipelines.shell.bash import BashTerminal

class CreateFileAndDirectory(batch.BatchQ):
    _ = batch.WildCard()
    directory = batch.Property()
    command = batch.Property()

    terminal = batch.Controller(BashTerminal)

    create_dir = batch.Function() \
        .home().chdir(_) \
        .exists(directory).Qdon(1).mkdir(directory, True) \
        .chdir(directory)

    create_file = batch.Function(create_dir) \
        .send_command(command)
```

Example: Short version

While the previous code is pretty taking into consideration what it does it can still be made shorter. One of the neat features of function queues are that you can predefine certain patterns - as for instance creating a directory in your home directory:

```
from batchq.core import batch
from batchq.pipelines.shell.bash import BashTerminal
from batchq.shortcuts.shell import home_create_dir, send_command

class CreateFileShort(batch.BatchQ):
    _ = batch.WildCard()
    directory = batch.Property()
    command = batch.Property()

    terminal = batch.Controller(BashTerminal)

    create_dir = home_create_dir(directory,_)
    create_file = send_command(command, inherits = create_dir)
```

Overwriting Controllers

A key feature of BatchQ is that you can overwrite controllers from your previous models. This means that once you have made a model for one pipeline you can use it on another one by simply overwriting the controller. In the following we overwrite the controller for one pipeline with a new controller for another pipeline:

```
from batchq.core import batch

class Pipe(object):
    def hello(self, msg):
        print msg

class Model1(batch.BatchQ):
    ctrl = batch.Controller(Pipe)
    fnc = batch.Function(verbose=True).hello("Hello from FNC")

class ReplacementPipe(object):
    def hello(self, msg):
        print msg[::-1]

class Model2(Model1):
    ctrl = batch.Controller(ReplacementPipe)

Model1().fnc()
Model2().fnc()
```

The replacement pipeline reverses the string and the code produces the following output

```
Hello from FNC
CNF morf olleH
```

Example: CreateFileSSH

Clearly, the previous example comes in handy when we want to extent our script to support SSH:

```
from batchq.pipelines.shell.ssh import SSHTerminal
import getpass

class CreateFileSSH(CreateFileShort):
    server = batch.Property()
    username = batch.Property()
    password = batch.Property()
    terminal = batch.Controller(SSHTerminal, server, username, password)

user = raw_input("Username:")
pasw = getpass.getpass()
instance = CreateFileSSH("Documents/DEMO_SSH", "echo Hello SSH > hello.txt",
↪"localhost",user,pasw)
instance.create_file()
```

The three new properties are appended to the constructor arguments.

5.2 Tutorial: nohup Remote Submission

In this tutorial we are going to write a submission module using nohup. Actually, we will not use nohup itself as this is a rather unstable application, but instead we will use the bash equivalent `([command] )` as this is a much more stable method.

5.2.1 Basic functionality

By now we have already written the first many small classes using BatchQ and therefore, the only thing we really need to know is which parameters the class should depend on and which methods we should implement.

A nohup module should take a command as input parameter as well as a working directory. It should implement the methods startjob, isrunning and clean. Subsequently, these function would need a function that enters the working directory and a function that checks whether a process is running. The implementation is straight forward:

```
from batchq.core.library import Library
from batchq.core import batch
from batchq.pipelines.shell.bash import BashTerminal
from batchq.shortcuts.shell import home_create_dir, send_command

class NoHUPStart(batch.BatchQ):
    _r = batch.WildCard(reverse = True)
    _ = batch.WildCard()

    input_directory = batch.Property()
    working_directory = batch.Property()
    command = batch.Property()

    terminal = batch.Controller(BashTerminal)

    workdir = batch.Function() \
        .home().chdir(_).exists(working_directory) \
        .don(1).mkdir(working_directory).chdir(working_directory)

    _set_copy_dirs = batch.Function(workdir, verbose=False) \
        .pjoin(input_directory, "/*").pjoin(working_directory, "/").cp(_r, _r) \

    transfer_infiles = batch.Function(_set_copy_dirs , verbose=False)
        .cp(_r, _r).don(1).throw("Failed to transfer files.")

    transfer_outfiles = batch.Function(_set_copy_dirs, verbose=False) \
        .cp(_, _).don(1).throw("Failed to transfer files.")

    startjob = batch.Function(workdir) \
        .join("(", command, " > .bacthq.output & echo $! > .batchq.pid )").send_
↪command(_)

    transfer_startjob = batch.Function(transfer_infiles) \
        .call(startjob)


    getpid = batch.Function(workdir) \
        .cat(".batchq.pid")

    isrunning = batch.Function(get_pid) \
        .isrunning(_)
```

```
    wasstarted = batch.Function(workdir) \
        .exists(".batchq.output")

    log = batch.Function(wasstarted) \
        .do(1).cat(".batchq.output")

    clean = batch.Function(wasstarted) \
        .do(1).rm(".batchq.*", force = True)


Library.queues.register("nohup",NoHUPStart)
if __name__=="__main__":
    dir1 = raw_input("Enter a input directory: ")
    dir2 = raw_input("Enter a output directory: ")
    cmd = raw_input("Enter a command: ")
    x = NoHUPStart(dir1, dir2, cmd)

    while True:
        x.interact()
#    print "1) Transfer input files"
#    print "2) Submit job"
#    print "3) Transfer input and submit job"

#    print "4) Check if it was started"
#    print "5) Check if it is running"
#    print "6) Transfer output files"
#    print "7) Show log"
#    print "8) Clean up"

#    print "Q) Quit"
#    print ""
#    choice = raw_input("# :")
```

5.2.2 Full functionality

5.3 Tutorial: LSF Remote Submission

5.4 API Reference

5.4.1 Core Pipeline Classes

Process

BasePipe

VT100 Terminal Interpreter

5.4.2 Shell Pipelines

BatchQ comes with a number of predefined pipelines. Included in these are three shells, `BashTerminal`, `SSHTerminal` and `SFTPTerminal`, and one shell based class `FileCommander`.

Bash terminal

SSH terminal

SFTP terminal

FileCommander

5.4.3 Math Terminals

5.4.4 Batch Model

Indices and tables

- genindex
- modindex
- search