Computational and Systems Biology at Yale-NUS: Documentation

September 20, 2016

The following is a brief documentation of the basic functionality of the scobrapy package for the Yale-NUS Computational and Systems Biology team. Scobrapy was designed by Maurice Cheung as a supplementary package to the basic functionality of cobrapy, and thus this documentation is best read in conjunction with the cobrapy documentation.

Contents

1 Getting Started

1.1 Installation

1.1.1 Auto-installation

As of September 18th, 2016, code is available on the Computational and Systems Biology group for the auto-installation of the scobra package. This code has been tested for Windows 10 systems.

1.1.2 Manual installation

Scobra can also be installed manually:

1. Acquire the scobra directory from `https://github.com/mauriceccy/scobra` by direct download or through **git clone https://github.com/mauriceccy/scobra**.
2. Place the directory in the location you intend to keep it in. If you place the package in your home directory, you can skip the next step.
3. Update the path variable in python using the **sys.path.append(***path/to/scobrapy***)** command.

For example, here is a possible scobra installation (note: bash prompts use $, while python prompts are denoted >>>).

```
$ cd Home/Computational_biology
$ git clone https://github.com/mauriceccy/scobra
$ python
>>> import sys
>>> sys.path.append(Home/Computational_biology)
```

Now, when you wish to use scobra in your code, import it as you would any other package:

```
>>> import scobra
>>> from scobra.io import * [import all modules from subpackage io]
```

1.2 Updates

Scobra packages are updated on occasion with bug fixes or additional packages. In order to maintain your local copy of scobra, it is recommended that you regularly check for updates.

If you downloaded the scobra package using the automatic installation, or if you downloaded the file manually, you will need to download the updated scobra and overwrite your local version.

If you downloaded scobra using git, you can take advantage of git branch management to update your local copy of scobra in place. To do so:

1. Navigate to the scobra directory using your command line interface.
2. Use **git status** to check for updates.
3. Run **git pull** to update with the master branch.

Note: If you have made changes to the scobra directory locally, **git pull** will return a merge conflict. In this case, consult with the git documentation (`https://git-scm.com/documentation`) to resolve the merge conflict, then run the code above as normal.

2 The Essentials of Scobrapy

2.1 Load Model

Constraints based metabolic modelling tool 'scobra' can handle model files both in sbml and scrumpy formats. To load a model, type

```
>>> m=scobra.Model('PATH-OF-MODEL-FILE')
```

For example:

```
>>> m = scobra.Model('/home/user/toy.xml') [SBML model file]
>>> m = scobra.Model('/home/user/toy.spy','scrumpy') [ScrumPy model file]
```

2.2 Reactions

2.2.1 List of reactions

To obtain a list of reaction IDs, type:

```
>>> m.Reactions()
['R1', 'R2', 'R3', 'R4', 'R5', 'R6', 'R7', 'R8']

>>> len(m.Reactions()) [gives count of the reactions]
8
```

2.2.2 Delete reaction(s)

A single reaction can be removed using the command DelReaction. To remove multiple reactions, a list of reactions can be entered using the **DelReactions(**[*List of reaction IDs*]**)** command:

```
>>> m.DelReaction('R4')
>>> m.Reactions()
['R1', 'R2', 'R3', 'R5', 'R6', 'R7', 'R8']
```

2.2.3 Gene reaction association

To obtain either the reaction associated with a gene, or the genes associated with a reaction, use **GenesToReactionsAssociations** or **ReactionsToGenesAssociations**:

```
>>> m.GenesToReactionsAssociations {'Gr6': ['R6'], 'Gr7': ['R7'],
'Gr4': ['R4'], 'Gr5': ['R5'], ···}
>>> m.ReactionsToGenesAssociations
{'R4': ['Gr4'], 'R5': ['Gr5'], 'R6': ['Gr6'], 'R7': ['Gr7'], ···}
```

2.2.4 Obtain reaction names

To obtain reaction names one can use the command **GetReactionName(***Reaction ID***)**.

```
>>> m.GetReactionName('R1')
'R1'

>>> m.GetReactionNames(['R1','R2'])
['R1', 'R2']
```

2.2.5 *Reaction* object

We can get reactions as a *class* object in scobra that can be utilized for several other (useful) built-in *methods*. This can provide important information about the structure/function of the model. We can obtain a *reaction* class object and assign it to a variable, for example:

```
>>> r4=m.GetReaction('Reaction ID')
>>> r4,r5 = m.GetReactions(['R4','R5'])
```

Methods which can be used on a given class can be obtained using python's built-in function **dir()**:

```
>>> r4 = m.GetReaction('R4')
>>> dir(r4)
[list of attributes for the object r4]
```

2.2.6 Number of substances in a reaction - degree

We can obtain a Python dictionary containing keys for the reactions and respective values for the number of involved substances. For example:

```
>>> m.ReactionsDegree()
{'R4': 4, 'R5': 3, 'R6': 2, 'R7': 1, 'R1': 1, 'R2': 1, 'R3': 1, 'R8': 1}

>>> m.ReactionsDegree()['R5']
3
```

2.2.7 Print reactions

Model reactions can be printed out using the command **PrintReaction** or **PrintReactions**.

```
>>> m.PrintReaction('R5')
R5  B + 3.0 C <=> E
```

2.2.8 Reaction to sub-systems association

We can obtain the part(s) of the metabolism in which specific reactions are involved using the method **ReactionsToSubsystemsAssociations**. The reverse is obtained using the **SubsystemsToReactionsAssociations** method. This method is applicable to model object created with sbml model format (m). Presently, ScrumPy model format does not support localization of a reactions in the model file.

```
>>> m.ReactionsToSubsystemsAssociations
{'R4': ['XR4_Metabolism'], 'R5': ['XR5_Metabolism'], ···}
```

```
>>> scrumpymodel.ReactionsToSubsystemsAssociations
{'R4': [], 'R5': [], 'R6': [], 'R7': [], 'R1': [], 'R2': [], 'R3': [], 'R8': []}

>>> m.SubsystemsToReactionsAssociations [for subsystem to reaction association]
{'XR3_Metabolism': ['R3'], 'XR7_Metabolism': ['R7'], ···}
```

2.2.9 Add reactions to scobra model

Reaction(s) can be added to the scobra model using the method **AddReaction**. Arguments in the method should be passed to describe a reaction. Reversible and irreversible reaction can be described by the third argument.

For example, new reaction 'R9' is added to the model object that includes reactants E and C (stoichiometric coefficient = -1 and -2, respectively) and product F (stoichiometric coefficient = 1).

```
>>> m.AddReaction('R9',{'E':-1,'C':-2,'F':1})
>>> m.PrintReaction('R9')
R9  2 C + E -- > F [irreversible]

>>> m.DelReactions(['R9'])
>>> m.AddReaction('R9',{'E':-1,'C':-2,'F':1},True)
>>> m.PrintReaction('R9')
R9  2 C + E <=> F [reversible]
```

Multiple reactions are easily created using loops and lists with tuples.

2.2.10 Change of reaction stoichiometry

Existing reaction's stoichiometry can be changed using **ChangeReactionStoichiometry**. Note that this method can also update a reaction by changing reactants and/or products (with an existing set of metabolites).

```
>>> m.ChangeReactionStoichiometry('R9',{'E':-1,'C':-3,'F':2})
>>> m.PrintReaction('R9')
```

```
R9  3 C + E <=> 2 F

>>> m.ChangeReactionStoichiometry('R9',{'E':-1,'C':3,'F':-2})
>>> m.PrintReaction('R9')
R9  E + 2 F <=> 3 C

>>> m.ChangeReactionStoichiometry('R9',{'E':-1,'C':3,'F':-2,'A':1})
>>> m.PrintReaction('R9')
R9  E + 2 F <=> A + 3 C
```

2.3 Metabolites

2.3.1 Get metabolite

To get a list of metabolites present in the scobra model use the following command:

```
>>> m.Metabolites()
['A', 'B', 'C', 'D', 'E', 'F']
```

To get metabolite class object(s) use **GetMetabolite(***Metabolite***)** or **GetMetabolites([***list of metabolites***])**.

```
>>> met=m.GetMetabolite('A')
>>> met.name
'A_internal'
```

Various parcels of information are now available regarding the metabolite:

```
>>> met.reaction(s)
frozenset([<Reaction R1 at 0xa9f8b4c>, <Reaction R4 at 0xa9f8ccc>])

>>> met.compartment
'internal'
```

```
>>> met.charge [charge]
-2

>>> metA, MetB = m.GetMetabolites(['A','B'])
```

If you only need the name, and not the object, use **GetMetaboliteName(s)**.

```
>>> m.GetMetaboliteName('A')
'A'

>>> m.GetMetaboliteNames(['A','B'])
['A', 'B']
```

2.3.2 Dead, Dead End and Peripheral Metabolites

Dead end metabolites

Dead end metabolites in the internal stoichiometric model present in the periphery can be identified by **DeadEndMetabolites**. Removal of a dead end metabolite sometime marks another metabolite as dead.

```
>>> m.DeadEndMetabolites()
['F']
```

Dead metabolites

Dead metabolites are those which either produced or consumed by a reaction. Using **DeadMetabolites** one can find all metabolites that are not involved in any allowed reactions.

```
>>> m.DeadMetabolites()
['F']
```

The distinction between dead end metabolites and dead metabolites is clarified with the addition a successive dead reaction that utilizes 'F' as a reactant:

```
>>> m.AddReaction('R9',{'F':-1,'G':1})
>>> m.DeadMetabolites()
['G', 'F']

>>> m.DeadEndMetabolites()
['G']
```

Peripheral metabolites

Instructions are shown with temporary addition of reaction R9.

```
>>> m.PeripheralMetabolites("all")
['F']

>>> m.AddReactions('R9', {'F':1, 'G':-1})
>>> m.PrintReactions
R1    -> A
R2    -> B
R3    -> C
R4    A + B + C -> D
R5    B + 3.0 C <=> E
R6    E -> F
R7    D ->
R8    E ->
R9    G -> F

>>> m.PeripheralMetabolites("all")
['G', 'F']

>>> m.PeripheralMetabolites("Produced")
['F']

>>> m.PeripheralMetabolites("Consumed")
['G']

>>> m.PeripheralMetabolites("Orphan")
['G']
```

2.3.3 Blocked metabolites

Metabolites not involved in allowed reactions during flux analysis.

```
>>> m.BlockedMetabolites()
['F']
```

2.3.4 Metabolites involved in reactions - degree

The degree of a metabolite can be seen using the **MetaboliteDegree(s)** method:

```
>>> m.MetabolitesDegree(['B','F'])
{'B': 3, 'F': 1}

>>> m.MetabolitesDegree('B')
{'B': 3}
```

2.3.5 Delete metabolites

Metabolites can be deleted using the **DelMetabolites** method:

```
>>> m.Metabolites()
['A', 'B', 'C', 'D', 'E', 'F']

>>> m.DelMetabolite('A')
>>> m.Metabolites()
['B', 'C', 'D', 'E', 'F']

>>> m.DelMetabolites(['B','C'])
>>> m.Metabolites()
['D', 'E', 'F']
```

2.3.6 Produce metabolites

Metabolites those are either produced or not produced can be obtained using **ProduceMetabolites**.

```
>>> m.ProduceMetabolites()
{'Produced': ['A', 'B', 'C', 'D', 'E', 'F'], 'Not Produced': ['G']}
```

2.4 Genes

2.4.1 Get genes

A list of genes involved in a model:

```
>>> m.Genes()
['Gr1', 'Gr2', 'Gr3', 'Gr4', 'Gr5', 'Gr6', 'Gr7', 'Gr8']
```

2.4.2 Genes to metabolism

Similar to the method for reactions, a list of gene subsystems can be obtained using **GenesToSubsystemsAssociations** or **SubsystemsToGenesAssociations**. For example,

```
>>> m.GenesToSubsystemsAssociations
{'Gr6': ['XR6_Metabolism'], 'Gr7': ['XR7_Metabolism'], ···}

>>> m.SubsystemsToGenesAssociations
{'XR3_Metabolism': ['Gr3'], 'XR7_Metabolism': ['Gr7'], ···}
```

2.4.3 Get gene name

If you only need the name of a gene and not the gene object, use the **GetGeneName** method:

```
>>> m.GetGeneName('R1')
'R1'

>>> m.GetGeneNames(['R1','R2'])
['R1', 'R2']
```

2.4.4 Get gene class object

Gene objects also exist within the model, and can be called and assigned a variable.

```
>>> g1=m.GetGene('Gr1')
>>> g1,g2=m.GetGenes(['Gr1','Gr2'])
```

2.5 Reaction and metabolite involvements

Metabolites and reaction objects are obviously associated, and these associations can be called using **InvolvedWith**:

```
>>> m.InvolvedWith('A')
{<Reaction R4 at 0xa3ebf4c>: -1.0, <Reaction R1 at 0xa3ebd4c>:
1.0}

>>> m.InvolvedWith('R1')
{<Metabolite A at 0xa3ebacc>: 1.0}
```

2.6 Neighbours in the Metabolic Graph

Neighbours are immediately adjacent reactions or metabolites; in other words, a metabolites neighbour are the metabolites which share a reaction, while a reactions neighbours are those reactions which consume its product or produce its reactants. Neighbours can be called using the **GetNeighbours** or **GetNeighboursAsDic** methods.

```
>>> m.GetNeighbours('A')
['C', 'B', 'D']

>>> m.GetNeighbours('R1')
['R4']

>>> m.GetNeighboursAsDic('R3')
{'C': ['R4', 'R5']}
```

3 Flux Analysis

3.1 Flux Balance Analysis (FBA)

Scobra models can be used to simulate metabolism using flux balance analysis. For the sake of simplicity, all simulations here use the simple toy model shown in this figure:

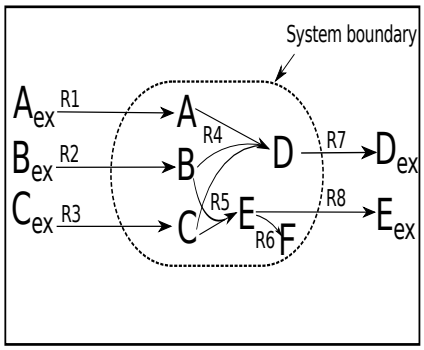

Figure 1: Simple model

```
m = scobra.Model('/home/user/toy.xml')
```

3.1.1 Set flux bounds to the reactions

Upper and lower limit of a reaction can be set using **SetConstraint** for a single reaction or **SetConstraints(**{*dict of reactions and flux bounds*}).

```
>>> m.SetConstraint('R1',0,10)
>>> m.SetConstraints({'R2':(0,13),'R3':(0,30)})
```

3.1.2 Set reaction ratios

Constraints can also be applied using the **SetReactionFixedRatio** method. This constrains the ratio of flux between two or more reactions, and can be very useful for building a set of constraints. The method accepts as an argument a dictionary where they keys are reactions and the values are the respective flux ratios. For example,

```
>>> m.SetReactionFixedRatio({ R1:1.5, R2: 2.2, R3: 0, ··· })
```

3.1.3 Set objective

Suppose the objective is to calculate maximum growth rate in R7 and R8, i.e., maximizing *Dex* and *Eex* production. Objective reactions (reactions to minimize or maximize) and theire maximization or minimization can be set using **SetObjective** and **SetObjDirec**, respectively.

```
>>> m.SetObjective(['R7','R8'])
>>> m.SetObjDirec("Max")
```

3.1.4 Solving the Linear Programming (LP) problem

The constraints based LP problem can be solved using **Solve()**. For this example, the method returns an optimal growth rate, i.e., a feasible solution. For problems which are not possible to solve, **Solve()** will give return "infeasible":

```
>>> m.Solve()
optimal
```

3.1.5 Solutions

To get the flux solution as a dictionary use **GetSol**.

```
>>> m.GetSol()
{'R4': 10.0, 'R5': 3.0, 'R7': 10.0, 'R1': 10.0, 'R2': 13.0, ···}
```

3.1.6 Printing the solution

Flux distributions can also be obtained using **PrintSol**. For our example, the maximum growth rate (maximum possible flux under the input flux assumptions) for R7 and R8 are 10.0 and 3.0, respectively.

```
>>> m.PrintSol()
R3: 19.0
R2: 13.0
R7: 10.0
R4: 10.0
R1: 10.0
R8: 3.0
R5: 3.0
```

3.1.7 Fixing the flux

Flux through a reaction can be fixed to a specific value using **SetFixedFlux**.

```
>>> m.SetFixedFlux('R1':0)
```

3.1.8 Get sum of fluxes

Get the sum of fluxes for a metabolite using **FluxSum(***Metabolite***)**.

```
>>> m.FluxSum('E')
3.0

>>> m.FluxSum('B')
13.0

>>> m.FluxSum('C')
19.0
```

3.1.9 Get constraints

In order to manipulate the behaviour of the plants, certain constraints are applied which limit the direction and extent of a specific reaction. Use **GetConstraint(s)** to view these constraints.

```
>>> m.GetConstraint('R1')
(0, 10)

>>> m.GetConstraints()
{'R4': (0.0, 1000.0), 'R5': (-1000.0, 1000.0), 'R6': (0.0, 1000.0), ···}
```

3.1.10 Summary information

The basic information of a solution is obtained using the **GetState** method:

```
>>> m.GetState()
{'objective_direction': 'maximize',
'solver': None,
'bounds': 1000.0,
'solution': <Solution 13.00 at 0xafdc7cc>,
'objective': {'R4': 0.0, 'R5': 0.0, 'R6': 0.0, 'R7': 1, 'R1': 0.0, 'R2': 0.0,
'R3': 0.0, 'R8': 1},
'quaduatic_component': None,
'constraints': {'R4': (0.0, 1000.0), 'R5': (-1000.0, 1000.0), ···}}
```

3.1.11 Get flux range

```
>>> m.AllFluxRange()
'R4': (0.0, 10.0), 'R5': (0.0, 10.0), 'R6': (0.0, 0.0), 'R7': (0.0, 10.0),
'R1': (0.0, 10.0), 'R2': (0.0, 13.0), 'R3': (0.0, 30.0), 'R8': (0.0, 10.0)

>>> m=scobra.Model('/home/user/toy.xml')
>>> m.AllFluxRange()
{'R4': (0.0, 1000.0), 'R5': (0.0, 333.33), 'R6': (0.0, 0.0), ···}
```

Note that the flux range is not equivalent to the constraints:

```
>>> m.GetConstraints()
{'R4': (0.0, 1000.0), 'R5': (-1000.0, 1000.0), 'R6': (0.0, 1000.0), ···}
```

3.2 Further Analysis

3.2.1 Flux Variability Analysis (FVA)

When scobra returns a flux solution for a given model and set of parameters, this solution is rarely unique. Rather, each reaction can sustain a variety of fluxes as the other fluxes vary. The variation of possible fluxes through a reaction can be important information regarding the importance of different reactions within a particular solution: smaller variation suggests greater importance.

In order to determine these different fluxes, we use the method **FVA**, with arguments for the reactions for which you wish to see the flux variation (as a list) and the number of processes you would like to run.

The example below returns flux variability for the first 3 reactions in the model *m*odel. It uses four processors to run the computation:

```
>>> model.FVA(model.Reactions()[0:3], processes=4)
```

The second example returns flux variability for the reactions in the list, using two processors:

```
>>> model.FVA(reaclist=['R4', 'R5'], processes=2)
```

4 The Server: Setting up the Machine

4.1 Installing Python and PIP using Homebrew for OSX

Most computers come installed with python 3.4 preinstalled. However, for modelling purposes we will be using python 2.7, which will need to be installed with careful attention to search paths, dependencies, etc. The easiest way to do this is to use Homebrew, a package manager for OSX.

What makes Homebrew so powerful is that it works to install packages using dependencies already on your computer. Further, Homebrew keeps track of your packages for you in a special folder, so it is less likely that the $PATH variable fails. Finally, Homebrew makes uninstalling very easy in cases something goes wrong. To install Homebrew, navigate to "http://brew.sh" and follow the instructions.

To install python and pip, use

```
>>>  brew install python
```

4.2 Installing Python and PIP on Windows (Alternative method for OSX)

Windows machines do not have Brew. Instead, download the Python2.7 MIS file from www.python.org. Run this file, taking note of the location in which python is being installed. After installation, navigate to control panel → System → Advanced → Environment Variables, and add a universal variable \ path\ to \ python.exe.
Download the get-pip.py file, found in the pip documentation at https://pip.pypa.io/en/stable/installing/, again taking note of where this is being downloaded. Navigate to this directory in the command line. Once in the correct directory, run

```
>>>  python get-pip.py
```

4.3 Installing Python Packages using PIP

Now that python is set up and pip is installed, you can begin to install the various packages you will be using for modelling. Recommended packages include:

- python-libsbml
- scipy
- numpy
- cobra
- lxml
- ipython

To use pip to install a package, use:

```
>>>  pip install package
```

where package is replaced by the name of the package you wish to install. Note that many of the names pip uses to reference packages are not identical to those you might use to refer to a package.

5 The Server: Basics

5.1 Navigating from the command line

In order to control the server, you will need to know how to navigate from the command line interface. The server is a Linux Redhat system; as such, its commands are consistant with Some basic commands for navigation follow:

- pwd
- cd
- ls
- mkdir
- rm

5.2 Starting SSH with UNIX systems

The ssh command is used in the command line interface in order to establish a secure connection with the remote machine.

```
>>>  ssh username@172.25.20.52
```

When prompted, enter your password. You are now logged into the server, and will remain logged in until you exit the session using the exit command.

```
>>>  exit
```

5.3 Starting SSH with Windows systems

Download PuTTY (http://www.putty.org) and use the desktop client to connect to the server at IP 172.25.20.52

5.4 Transferring Files to Remote Machine on Windows

To transfer files, install the WinSCP program from winscp.com and use the GUI provided.

5.5 Transferring Files to Remote Machine on OSX

Unlike Windows, UNIX system users will use the CLI to transfer files (a GUI can be installed, but is unnecessary). To move files between a local and remote machine, use the sftp command.

```
>>>  sftp username@127.25.20.52
```

This command opens an sftp interface in the command line. The sftp interface has a basic subset of the functionality of an ssh connection.

Navigational commands will automatically be run on the remote machine, but the local machine can still be navigated be preceding all commands

with l, e.g. lls, lpwd, etc. To temporarily direct all commands to the local machine, use the ! command.

```
>>>  !
```

To return to the remote machine, use the exit command.

```
>>>  exit
```

To move files from the local working directory to the remote working directory, utilize the put command.

```
>>>  put localFile
```

To move files from the remote working directory to the local working directory, utilize the get command.

```
>>>  get remoteFile
```

To move directories, use the put and get commands along with the -r tag.

```
>>>  get -r remoteDirectory
```

Note that the -r tag has outstanding bugs, e.g. an empty directory of the same name as the transfer directory must exist in the working directory of the receiving machine.

When the session is finished, stop the sftp:

```
>>>  bye
```

5.6 Group Account

Currently, there is a group account that can be used to share code for others to look at and use. This account has the following details:

Username: group
Password: [on request from Maurice or Markus]

6 The Server: Advanced

6.1 Remote Access

In order to access the server while abroad, it is necessary to use a Virtual Private Network. To access the correct VPN, download and install the forticlient program. The remote gateway SoC VPN can be found at: *webvpn.comp.nus.edu.sg*.

6.2 Running local files remotely

You may desire to run local scripts remotely. Unfortunately, local scripts will not be interactive when they are run remotely. For this reason, we recommend copying the files to the remote machine before running. However, it is possible to run local files on a remote machine. The simplest method is to pipe any script on the local machine to the remote machine (replace python with the desired program, if you don't want to execute the script using python):

```
>>>  cat /path/to/file — ssh username@172.25.20.52 python -
```

6.3 Continuous Access Using Screen

In the event that it is necessary to run a extended calculation, you will make use of a powerful program called screen. To start screen, simply type:

```
>>>  screen
```

Screen is used to run multiple sessions over the same ssh connection. More importantly, screen allows the user to run commands while disconnected from the server. Screen has an extensive navigational syntax; to learn more:

```
press ctrl + a, then ?
```

Run processes as normal. When you wish to detach from screen:

```
press ctrl + a, then d
```

To reattach:

```
>>>  screen -r
```

To lock your session, use:

```
press ctrl + a, then x
```

To terminate your screen session:

```
>>>  screen -r
```

or

```
press ctrl + a, then k
```

6.4 Graphic User Interface

Download the GUI from https://www.realvnc.com/download/viewer/.

6.5 Jupyter/iPython Notebook for OSX

While most users will access the server through the CLI or the GUI, it may be of interest that the server can be controlled using jupyter notebook. This requires the installation of the jupyter tools on the local machine. After installation, access the server using the CLI. Start a notebook on the remote machine:

```
>>>  sudo jupyter notebook - -no-browser
```

In a new tab in the CLI on the local machine:

```
>>>  ssh -NL 8157:localhost:8888 username@172.25.20.52
```

Now you can navigate to localhost:8157 on your browser to use the notebook.

When using the group server, port 8888 may be occupied, blocking access. In this case, it is necessary to host through a different port. To do so, replace 8888 with a new port number. In general, any port less than 1024 or greater than 49151 will be reserved. Further, some ports may not work for our purposes. Trial and error, however, will quickly lead to an open port.
To exit ipython notebook,

```
>>>  ctrl + c
```

6.6 Jupyter/iPython Notebook for Windows

To open a Jupyter Notebook on Windows:

1. Open PuTTy
2. Enter the IP of the server
3. Navigate to Tunnels under SSH
4. Add a new forwarding port where the source-port is 8157 and destination is localhost:8888
5. Navigate back and save these settings for later use
6. Open connection
7. Navigate to localhost:8157 on browser