SCOOP Documentation

Release 0.5.3

Marc Parizeau, Olivier Gagnon, Marc-André Gardner, Yannick Hold

August 28, 2012

CONTENTS

SCOOP (Scalable COncurrent Operations in Python) is a distributed task module allowing concurrent parallel programming on various environments, from heterogeneous grids to supercomputers.

Our philosophy is based on these ideas:

- The **future** is parallel;
- **Simple** is beautiful;
- **Parallelism** should be simpler.

These tenets are translated concretely in a minimum number of functions allowing maximum parallel efficiency while keeping at minimum the inner knowledge required to use them. It is implemented with Python 3 in mind while being compatible with 2.6+ to allow fast prototyping without sacrificing efficiency and speed.

CHAPTER

ONE

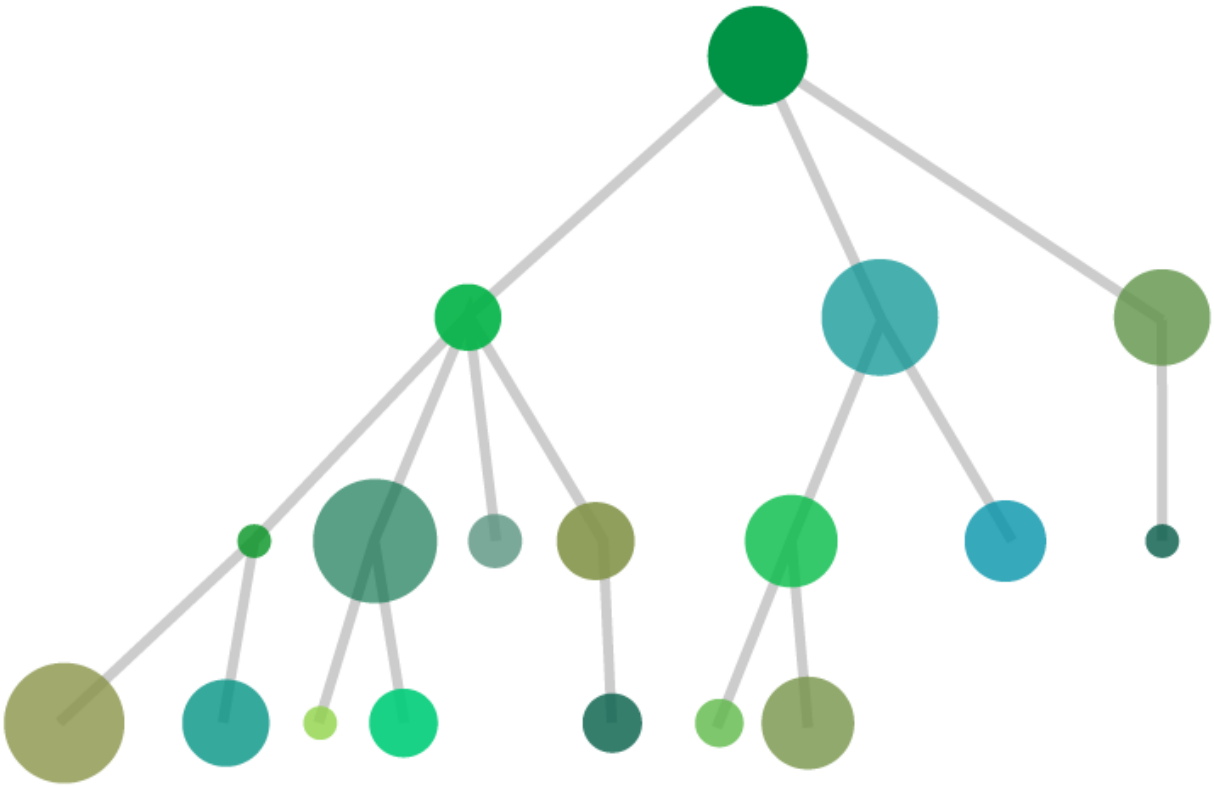
FEATURES

SCOOP has many features and advantages over Futures, multiprocessing and similar modules, such as:

- Harness the power of **multiple computers** over network;
- Ability to spawn subtasks within tasks;
- API compatible with **PEP 3148**;
- Parallelizing serial programs with only minor modifications;
- Efficient load-balancing.

1.1 Anatomy of a SCOOPed program

SCOOP can handle multiple diversified multi-layered tasks. You can submit your different functions and data simultaneously and effortlessly while the framework executes them locally or remotely. Contrarily to most multiprocessing frameworks, it allows to launch subtasks within tasks.



Through SCOOP, you can simultaneously execute tasks that are of different nature (Discs of different colors) or different by complexity (Discs radiuses). The module will handle the physical considerations of parallelization, such as task distribution over your resources (load balancing), communications, etc.

1.2 Applications

The common applications of SCOOP consist of, but is not limited to:

- Evolutionary Algorithms
- Monte Carlo simulations
- Data mining
- Data processing
- I/O processing
- Graph traversal

CHAPTER

TWO

MANUAL

2.1 Install

2.1.1 Requirements

The software requirements for SCOOP are as follows:

- Python >= 2.6 or >= 3.2
- Greenlet >= 0.3.4
- PyZMQ and libzmq >= 2.2.0
- **ssh** for remote execution

2.1.2 Installation

To install SCOOP and its other dependencies, use pip as such:

```
pip install scoop
```

Note: If you don't already have libzmq installed in a default library location, please visit the PyZMQ installation page for assistance.

Remote usage

Because remote host connection needs to be done without a prompt, you must use ssh keys to allow passwordless authentication. You should make sure that your public ssh key is contained in the `~/.ssh/authorized_keys2` file on the remote systems (Refer to the ssh manual). If you have a shared `/home/` over your systems, you can do as such:

```
[~]$ mkdir ~/.ssh; cd ~/.ssh
[.ssh]$ ssh-keygen -t dsa
[.ssh]$ cat id_dsa.pub >> authorized_keys2
```

Note: If your remote hosts needs special configuration (non-default port, some specified username, etc.), you should do it in your ssh client configuration file (by default `~/.ssh/config`).

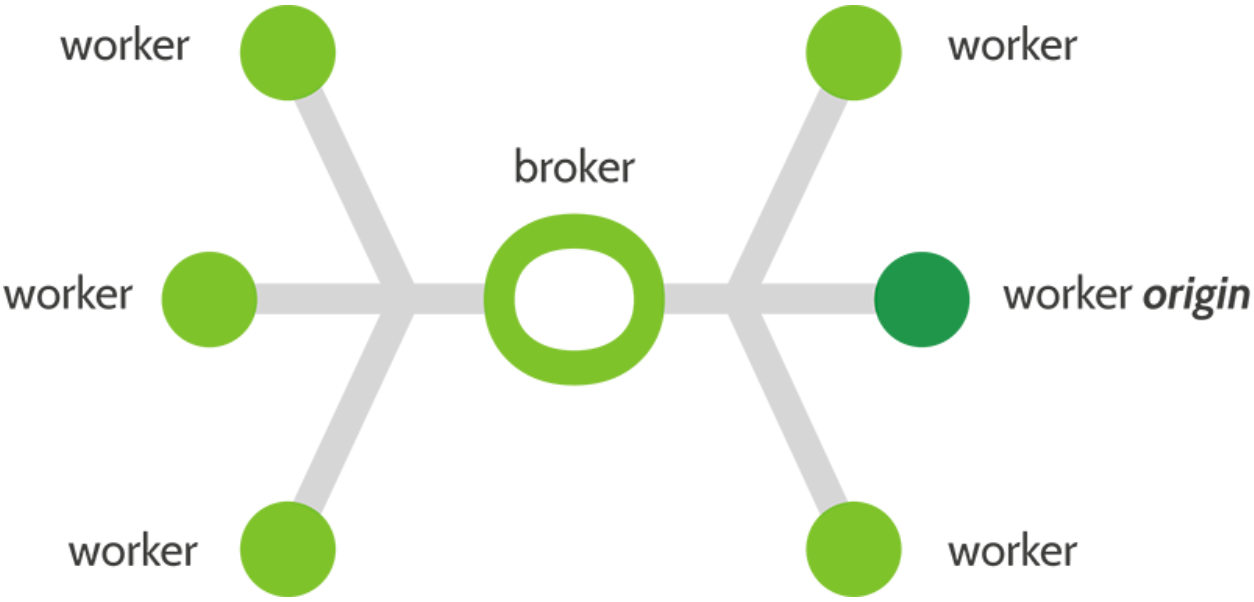
2.2 Usage

2.2.1 Nomenclature

Keyword	Description
Future(s)	The Future class encapsulates the asynchronous execution of a callable.
Broker	Process dispatching Futures.
Worker	Process executing Futures.
Root	The worker executing the root Future.

2.2.2 Architecture diagram

The future(s) distribution over workers is done by a variation of the Broker pattern. In such a pattern, workers act as independant elements that interact with a broker to mediate their communications.



2.2.3 How to use SCOOP in your code

The philosophy of SCOOP is loosely built around the *futures* module proposed by **PEP 3148**. It primarily defines a `map()` and a `submit()` function allowing asynchroneous computation that SCOOP will propagate to its workers.

Map

A *map()* function applies multiple parameters to a single function. For example, if you want to apply the *abs()* function to every number of a list:

```
import random
data = [random.randint(-1000,1000) for r in range(1000)]

# Without Map
result = []
for i in data:
  result.append(abs(i))
```

```
# Using a Map
result = list(map(abs, data))
```

SCOOP's `map()` returns a generator retriving the results in their proper order. It can thus act as a parallel substitute to the standard *map()*, for instance:

```
# Script to be launched with: python -m scoop scriptName.py
import random
from scoop import futures
data = [random.randint(-1000,1000) for r in range(1000)]

if __name__ == '__main__':
    # Python's standard serial function
    dataSerial = list(map(abs, data))

    # SCOOP's parallel function
    dataParallel = list(futures.map(abs, data))

    assert dataSerial == dataParallel
```

Warning: In your root program, you *must* check `if __name__ == __main__` as show above. Failure to do so will result in every worker trying to run their own instance of the program. This ensures that every worker waits for parallelized tasks spawned by the root worker.

Note: Your callable function passed to SCOOP must be picklable in its entirety.

Submit

SCOOP's `submit()` returns a `Future` instance. This allows a finer control over the Futures, such as out-of-order results retrieval.

Note: Functions submited to scoop must return a value. Keep in mind that objects are not shared between workers and that changes made to an object in a function are not made in every workers.

2.2.4 Examples

Examples are available in the `examples/` directory of SCOOP.

Please refer to the *Examples* page where detailed explanations are available.

2.2.5 How to launch SCOOP programs

The scoop module spawns the needed broker and workers on a given list of computers, including remote ones via **ssh**.

Programs using SCOOP need to be launched with the *`-m scoop`* parameter passed to Python, as such:

```
cd scoop/examples/
python -m scoop fullTree.py
```

Note: When using a Python version prior to 2.7, you must start SCOOP using *`-m scoop.__main__`*.

You should also consider using an up-to-date version of Python.

Here is a list of the parameters that can be passed to SCOOP:

```
python -m scoop --help
usage: python -m scoop [-h]
                       [--hosts [HOSTS [HOSTS ...]] | --hostfile HOSTFILE]
                       [--path PATH] [--nice NICE]
                       [--verbose] [--log LOG] [-n N]
                       [-e] [--broker-hostname BROKER_HOSTNAME]
                       [--python-executable PYTHON_EXECUTABLE]
                       [--pythonpath PYTHONPATH] [--debug]
                       executable ...

Starts a parallel program using SCOOP.

positional arguments:
  executable            The executable to start with SCOOP
  args                  The arguments to pass to the executable

optional arguments:
  -h, --help            show this help message and exit
  --hosts [HOSTS [HOSTS ...]], --host [HOSTS [HOSTS ...]]
                        The list of hosts. The first host will execute the
                        root program. (default is 127.0.0.1)
  --hostfile HOSTFILE   The hostfile name
  --path PATH, -p PATH  The path to the executable on remote hosts (default
                        is local directory)
  --nice NICE           *nix niceness level (-20 to 19) to run the executable
  --verbose, -v         Verbosity level of this launch script (-vv for more)
  --log LOG             The file to log the output (default is stdout)
  -n N                  Total number of workers to launch on the hosts.
                        Workers are spawned sequentially over the hosts.
                        (ie. -n 3 with 2 hosts will spawn 2 workers on the
                        first host and 1 on the second.) (default: Number of
                        CPUs on current machine)
  -e                    Activate ssh tunnels to route toward the broker
                        sockets over remote connections (may eliminate routing
                        problems and activate encryption but slows down
                        communications)
  --broker-hostname BROKER_HOSTNAME
                        The externally routable broker hostname / ip (defaults
                        to the local hostname)
  --python-executable PYTHON_EXECUTABLE
                        The python executable with which to execute the script
  --pythonpath PYTHONPATH
                        The PYTHONPATH environment variable (default is
                        current PYTHONPATH)

  --debug               Turn on the debuging
```

A remote workers example may be as follow:

```
python -m scoop --hostfile hosts -vv -n 6 your_program.py [your arguments]
```

Argument	Meaning
-m scoop	**Mandatory** Uses SCOOP to run program.
–hostfile	hosts is a file containing a list of host to launch SCOOP
-vv	Double verbosity flag.
-n 6	Launch a total of 6 workers.
your_program.py	The program to be launched.
[your arguments]	The arguments that needs to be passed to your program.

Note: Your local hostname must be externally routable for remote hosts to be able to connect to it. If you don't have a DNS properly set up on your local network or a system hosts file, consider using the *--broker-hostname* argument to provide your externally routable IP or DNS name to SCOOP. You may as well be interested in the *-e* argument for testing purposes.

Hostfile format

You can specify the hosts with a hostfile and pass it to SCOOP using the *--hostfile* argument. The hostfile should use the following syntax:

```
hostname_or_ip 4
other_hostname 5
third_hostname 2
```

The name being the system hostname and the number being the number of workers to launch on this host.

Using a list of host

You can also use a list of host with the *--host [...]* flag. In this case, you must put every host separated by a space the number of time you wish to have a worker on each of the node. For example:

```
python -m scoop --host machine_a machine_a machine_b machine_b your_program.py
```

This example would start two workers on *machine_a* and two workers on *machine_b*.

Choosing the number of workers

The number of workers started should be equal to the number of cores you have on each machine. If you wish to start more or less workers than specified in your hostfile or in your hostlist, you can use the *-n* parameter.

Note: The *-n* parameter overrides any previously specified worker amount.

If *-n* if less than the sum of workers specified in the hostfile or hostlist, the workers are launched in batch by host until the parameter is reached. This behaviour may ignore latters hosts.

If *-n* if more than the sum of workers specified in the hostfile or hostlist, the remaining workers are distributed using a Round-Robin algorithm. Each host will increment its worker amount until the parameter is reached.

Be aware that tinkering with this parameter may hinder performances. The default value choosen by SCOOP (one worker by physical core) is generaly a good value.

2.2.6 Startup scripts (cluster or grid)

You must provide a startup script on systems using a scheduler. Here are some example startup scripts using different grid task managers. They are available in the `examples/submitFiles` directory.

Note: **Please note that these are only examples**. Refer to the documentation of your scheduler for the list of arguments needed to run the task on your grid or cluster.

Torque (Moab & Maui)

Here is an example of a submit file for Torque:

```
#!/bin/bash
## Please refer to your grid documentation for available flags. This is only an example.
#PBS -l procs=16
#PBS -V
#PBS -N SCOOPJob

# Path to your executable. For example, if you extracted SCOOP to $HOME/downloads/scoop
cd $HOME/downloads/scoop/examples

# Add any addition to your environment variables like PATH. For example, if your local python instal
export PATH=$HOME/python/bin:$PATH

# If, instead, you are using the python offered by the system, you can stipulate it's library path v
#export PYTHONPATH=$HOME/wanted/path/lib/python+version/site-packages/:$PYTHONPATH
# Or use VirtualEnv via virtualenvwrapper here:
#workon yourenvironment

# Launch SCOOP using the hosts
python -m scoop -vv fullTree.py
```

Sun Grid Engine (SGE)

Here is an example of a submit file for SGE:

```
#!/bin/bash
## Please refer to your grid documentation for available flags. This is only an example.
#$ -l h_rt=300
#$ -pe test 16
#$ -S /bin/bash
#$ -cwd
#$ -notify

# Path to your executable. For example, if you extracted SCOOP to $HOME/downloads/scoop
cd $HOME/downloads/scoop/examples

# Add any addition to your environment variables like PATH. For example, if your local python instal
export PATH=$HOME/python/bin:$PATH

# If, instead, you are using the python offered by the system, you can stipulate it's library path v
#export PYTHONPATH=$HOME/wanted/path/lib/python+version/site-packages/:$PYTHONPATH
# Or use VirtualEnv via virtualenvwrapper here:
#workon yourenvironment
```

```
# Launch the remotes workers
python -m scoop -vv fullTree.py
```

2.2.7 Pitfalls

Program scope

As a good Python practice (see **PEP 395**), you should always wrap the executable part of your program using:

```
if __name__ == '__main__':
```

This is mandatory when using parallel frameworks such as multiprocessing and SCOOP. Otherwise, each worker (or equivalent) will try to execute your code serially.

Evaluation laziness

The `map()` and `submit()` will distribute their Futures both locally and remotely. Futures executed locally will be computed upon access (iteration for the `map()` and `result()` for `submit()`).Futures distributed remotely will be executed right away.

Large datasets

Every parameter sent to a function by a `map()` or `submit()` gets serialized and sent within the Future to its worker. It results in slow speeds and network overload when sending large elements as a parameter to your function(s).

You should consider using a global variable in your module scope for passing large elements; it will then be loaded on launch by every worker and won't overload your network.

Incorrect:

```
from scoop import futures


def mySum(inData):
    """The worker will receive all its data from network."""
    return sum(inData)

if __name__ == '__main__':
    data = [[i for i in range(x, x + 1000)] for x in range(0, 8001, 1000)]
    results = list(futures.map(mySum, data))
```

Better:

```
from scoop import futures

data = [[i for i in range(x, x + 1000)] for x in range(0, 8001, 1000)]


def mySum(inIndex):
    """The worker will only receive an index from network."""
    return sum(data[inIndex])

if __name__ == '__main__':
    results = list(futures.map(mySum, range(len(data))))
```

SCOOP and greenlets

Warning: Since SCOOP uses greenlets to schedule and run futures. Programs that use their own greenlets won't work with SCOOP. However, you should consider replacing the greenlets in your code by SCOOP functions.

2.3 Examples

You can find the examples detailed on this page in the `examples/` directory of SCOOP.

Please check the *API Reference* for any implentation detail of the proposed functions.

2.3.1 Computation of π

A Monte-Carlo method to calculate π using SCOOP to parallelize its computation is found in `examples/piCalc.py`. You should familiarize yourself with Monte-Carlo methods before going forth with this example.

First, we need to import the needed functions as such:

Figure 2.1: Image from Wikipedia made by CaitlinJo that shows the Monte Carlo computation of π.

```
from math import hypot
from random import random
from scoop import futures
```

The Monte-Carlo method is then defined. It spawns two pseudo-random numbers that are fed to the hypot function which calculates the hypotenuse of its parameters. This step computes the Pythagorean equation ($\sqrt{x^2 + y^2}$) of the given parameters to find the distance from the origin (0,0) to the randomly placed point (which X and Y values were generated from the two pseudo-random values). Then, the result is compared to one to evaluate if this point is inside or outside the unit disk. If it is inside (have a distance from the origin lesser than one), a value of one is produced (red dots in the figure), otherwise the value is zero (blue dots in the figure). The experiment is repeated `tries` number of times with new random values.

The function returns the number times a pseudo-randomly generated point fell inside the unit disk for a given number of tries.

```
def test(tries):
    return sum(hypot(random(), random()) < 1 for i in range(tries))
```

One way to obtain a more precise result with a Monte-Carlo method is to perform the method multiple times. The following function executes repeatedly the previous function to gain more precision. These calls are handled by SCOOP using it's `map()` function. The results, that is the number of times a random distribution over a 1x1 square hits the unit disk over a given number of tries, are then summed and divided by the total of tries. Since we only covered the upper right quadrant of the unit disk because both parameters are positive in a cartesian map, the result must be multiplied by 4 to get the relation between area and circumference, namely π.

```
def calcPi(nbFutures, tries):
    expr = futures.map(test, [tries] * nbFutures)
    return 4. * sum(expr) / float(nbFutures * tries)
```

As *stated above*, you *must* wrap your code with a test for the __main__ name. You can now run your code using the command **python -m scoop**.

```
if __name__ == "__main__":
    print("pi = {}".format(calcPi(3000, 5000)))
```

2.3.2 Overall example

The `examples/fullTree.py` example holds a pretty good wrap-up of available functionnalities. It notably shows that SCOOP is capable of handling twisted and complex hierarchical requirements.

Getting acquainted with the previous examples is fairly enough to use SCOOP, no need to dive into this complicated example.

2.4 API Reference

2.4.1 Futures module

The following methods are part of the futures module. They can be accessed like so:

```
from scoop import futures

results = futures.map(func, data)
futureObject = futures.submit(func, arg)
...
```

More informations are available in the *Usage* document.

scoop.futures.**as_completed**(*fs*, *timeout=None*)

An iterator over the given futures that yields each as it completes.

Parameters

- **fs** – The sequence of Futures (possibly created by another instance) to wait upon.
- **timeout** – The maximum number of seconds to wait [To be done in future version of SCOOP]. If None, then there is no limit on the wait time.

Returns An iterator that yields the given Futures as they complete (finished or cancelled).

scoop.futures.**map**(*func*, **iterables*, ***kargs*)

Equivalent to map(func, *iterables, ...) but *func* is executed asynchronously and several calls to func may be made concurrently. The returned iterator raises a TimeoutError if *__next__()* is called and the result isn't available after timeout seconds from the original call to *map()* [To be done in future version of SCOOP]. If timeout is not specified or None then there is no limit to the wait time. If a call raises an exception then that exception will be raised when its value is retrieved from the iterator.

Parameters

- **func** – Any picklable callable object (function or class object with *__call__* method); this object will be called to execute the Futures. The callable must return a value.
- **iterables** – Iterable objects; each will be zipped to form an iterable of arguments tuples that will be passed to the callable object as a separate Future.
- **timeout** – The maximum number of seconds to wait [To be done in future version of SCOOP]. If None, then there is no limit on the wait time.
- **kargs** – A dictionary of additional keyword arguments that will be passed to the callable object.

Returns A generator of map results, each corresponding to one map iteration.

scoop.futures.**shutdown**(*wait=True*)
This function is here for compatibility with *futures* (PEP 3148).

Parameters **wait** – Unapplied parameter.

scoop.futures.**submit**(*func*, **args*, ***kargs*)
Submit an independent parallel scoop._types.Future that will either run locally or remotely as *func(*args, **kargs)*.

Parameters

- **func** – Any picklable callable object (function or class object with *__call__* method); this object will be called to execute the Future. The callable must return a value.
- **args** – A tuple of positional arguments that will be passed to the callable object.
- **kargs** – A dictionary of additional keyword arguments that will be passed to the callable object.

Returns A future object for retrieving the Future result.

On return, the Future is pending execution locally, but may also be transfered remotely depending on load or on remote distributed workers. You may carry on with any further computations while the Future completes. Result retrieval is made via the result() function on the Future.

scoop.futures.**wait**(*fs*, *timeout=None*, *return_when='ALL_COMPLETED'*)
Wait for the futures in the given sequence to complete.

Parameters

- **fs** – The sequence of Futures (possibly created by another instance) to wait upon.
- **timeout** – The maximum number of seconds to wait [To be done in future version of SCOOP]. If None, then there is no limit on the wait time.
- **return_when** – Indicates when this function should return. The options are:

FIRST_COMPLETED	Return when any future finishes or is cancelled.
FIRST_EXCEPTION	Return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to ALL_COMPLETED.
ALL_COMPLETED	Return when all futures finish or are cancelled.

Returns A named 2-tuple of sets. The first set, named 'done', contains the futures that completed (is finished or cancelled) before the wait completed. The second set, named 'not_done', contains uncompleted futures.

2.4.2 Future class

The submit() function returns a Future object. This instance possess the following methods.

class scoop._types.**Future**(*parentId*, *callable*, **args*, ***kargs*)
This class encapsulates an independent future that can be executed in parallel. A future can spawn other parallel futures which themselves can recursively spawn other futures.

add_done_callback(*callable*)
Attach a callable to the future that will be called when the future is cancelled or finishes running. Callable will be called with the future as its only argument.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises an Exception then it will be logged and ignored. If the callable raises another BaseException then behavior is not defined.

If the future has already completed or been cancelled then callable will be called immediately.

cancel()
If the call is currently being executed then it cannot be cancelled and the method will return False, otherwise the call will be cancelled and the method will return True.

cancelled()
True if the call was successfully cancelled, False otherwise.

done()
True if the call was successfully cancelled or finished running, False otherwise.

exception(*timeout=None*)
Return the exception raised by the call. If the call hasn't yet completed then this method will wait up to timeout seconds [To be done in future version of SCOOP]. If the call hasn't completed in timeout seconds then a TimeoutError will be raised. If timeout is not specified or None then there is no limit to the wait time.

If the future is cancelled before completing then CancelledError will be raised.

If the call completed without raising then None is returned.

Returns The exception raised by the call.

result(*timeout=None*)
Return the value returned by the call. If the call hasn't yet completed then this method will wait up to ''timeout" seconds [To be done in future version of SCOOP]. If the call hasn't completed in timeout seconds then a TimeoutError will be raised. If timeout is not specified or None then there is no limit to the wait time.

If the future is cancelled before completing then CancelledError will be raised.

If the call raised an exception then this method will raise the same exception.

Returns The value returned by the call.

running()
True if the call is currently being executed and cannot be cancelled.

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX