Noodles Documentation

Release 0.3.0

Johan Hidding, Felipe Zapata

Mar 21, 2019

Contents

CHAPTER 1

Introduction

Often, a computer program can be sped up by executing parts of its code *in parallel* (simultaneously), as opposed to *synchronously* (one part after another).

A simple example may be where you assign two variables, as follows `a = 2 * i` and `b = 3 * i`. Either statement is only dependent on `i`, but whether you assign `a` before `b` or vice versa, does not matter for how your program works. Whenever this is the case, there is potential to speed up a program, because the assignment of `a` and `b` could be done in parallel, using multiple cores on your computer's CPU. Obviously, for simple assignments like `a = 2 * i`, there is not much time to be gained, but what if `a` is the result of a time-consuming function, e.g. `a = very_difficult_function(i)`? And what if your program makes many calls to that function, e.g. `list_of_a = [very_difficult_function(i) for i in list_of_i]`? The potential speed-up could be tremendous.

So, parallel execution of computer programs is great for improving performance, but how do you tell the computer which parts should be executed in parallel, and which parts should be executed synchronously? How do you identify the order in which to execute each part, since the optimal order may be different from the order in which the parts appear in your program. These questions quickly become nearly impossible to answer as your program grows and changes during development. Because of this, many developers accept the slow execution of their program only because it saves them from the headaches associated with keeping track of which parts of their program depend on which other parts.

Enter Noodles.

Noodles is a Python package that can automatically construct a *callgraph* for a given Python program, listing exactly which parts depend on which parts. Moreover, Noodles can subsequently use the callgraph to execute code in parallel on your local machine using multiple cores. If you so choose, you can even configure Noodles such that it will execute the code remotely, for example on a big compute node in a cluster computer.

CHAPTER 2

Copyright & Licence

See http://www.esciencecenter.nl for more information on the NLeSC.

CHAPTER 3

Installation

Warning: We don't support Python versions lower than 3.5.

The core of Noodles runs on **Python 3.5** and above. To run Noodles on your own machine, no extra dependencies are required. It is advised to install Noodles in a virtualenv. If you want support for Xenon, install pyxenon too.

```
# create the virtualenv
virtualenv -p python3 <venv-dir>
. <venv-dir>/bin/activate

# install noodles
pip install noodles
```

Noodles has several optional dependencies. To be able to use the Xenon job scheduler, install Noodles with:

```
pip install noodles[xenon]
```

The provenance/caching feature needs TinyDB installed:

```
pip install noodles[prov]
```

To be able to run the unit tests:

```
pip install noodles[test]
```

CHAPTER 4

Documentation Contents

4.1 Eating noodles (user docs)

The primary goal of the *noodles* library is to ease the construction and execution of *computational workflows* using the Python language. This library is meant for scientists who want to perform complex compute-intensive tasks on parallel/distributed infrastructures in a readable, scalable and sustainable/reproducible? manner. A workflow is commonly modelled as a *directed acyclic graph* (DAG or simply graph) in which the computations are represented as nodes whereas the dependencies between them are represented as directed edges (indicating data transport).

4.1.1 A first example

Let's look at a small example of creating a diamond workflow, which consists of simple (arithmetic) functions:

```
from noodles import run_single
from noodles.tutorial import (add, sub, mul)

u = add(5, 4)
v = sub(u, 3)
w = sub(u, 2)
x = mul(v, w)

answer = run_single(x)

print("The answer is {0}.".format(answer))
```

That allmost looks like normal Python! The only difference is the *`run_single()`* statement at the end of this program. The catch is that none of the computation is actually done until the *`run_single()`* statement has been given. The variables `u`, `v`, `w`, and `x` only represent the *promise* of a value. The functions that we imported are wrapped, such that they construct the directed acyclic graph of the computation in stead of just computing the result immediately. This DAG then looks like this:

Running this program will first evaluate the result to `add(5, 4)`. The resulting value is then inserted into the empty slots in the depending nodes. Each time a node has no empty slots left, it is scheduled for evaluation. At the end, the

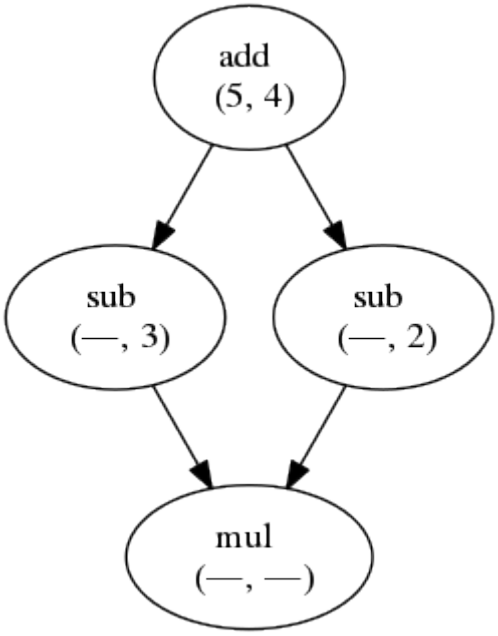


Fig. 1: The diamond workflow.

program should print:

```
The answer is 42.
```

At this point it is good to know what the module `noodles.tutorial` looks like. It looks very simple. However, you should be aware of what happens behind the curtains, to understand the limitations of this approach.

```
from noodles import schedule

@schedule
def add(a, b):
  """Adding up numbers! (is very uplifting)"""
  return a + b

@schedule
def sub(a, b):
  """Subtracting numbers."""
  return a - b

@schedule
def mul(a, b):
  """Multiplying numbers."""
  return a * b

...
```

The *`@schedule`* decorators takes care that the functions actually return *promises* instead of values. Such a *`PromisedObject`* is a placeholder for the expected result. It stores the workflow graph that is needed to compute the promise. When another *`schedule`*-decorated function is called with a promise, the graphs of the dependencies are merged to create a new workflow graph.

Note: The promised object can be of any type and can be used as a normal object. You access attributes and functions

of the object that is promised as you normally would. Be aware, however, that it is important to program in a functional way, so changing the attributes of a promised object is not a good idea. Instead, return a copy of the object with the changed values.

Doing things parallel

Using the Noodles approach it becomes very easy to paralellise computations. Let's look at a second example.

```
from noodles import (gather, run_parallel)
from noodles.tutorial import (add, sub, mul, accumulate)


def my_func(a, b, c):
    d = add(a, b)
    return mul(d, c)


u = add(1, 1)
v = sub(3, u)
w = [my_func(i, v, u) for i in range(6)]
x = accumulate(gather(*w))

answer = run_parallel(x, n_threads=4)

print("The answer is {0}, again.".format(answer))
```

This time the workflow graph will look a bit more complicated.

Here we see how a user can define normal python functions and use them to build a larger workflow. Furthermore, we introduce a new bit of magic: the *gather* function. When you build a list of computations using a list-comprehension like above, you essentially store a *list of promises* in variable `w`. However, schedule-decorated functions cannot easily see which arguments contain promised values, such as `w`, and which arguments are plain Python. The *gather* function converts the list of promises into a promise of a list, making it clear to the scheduled function this argument is a promise. The *gather* function is defined as follows:

```
@schedule
def gather(*lst):
    return lst
```

By unpacking the list (by doing `gather(*w)`) in the call to gather, each item in `w` becomes a dependency of the `gather` node in this workflow, as we can see in the figure above.

To make use of the parallelism in this workflow, we run it with *run_parallel()*. This runner function creates a specified number of threads, each taking jobs from the Noodles scheduler and returning results.

4.1.2 Running workflows

Noodles ships with a few ready-made functions that run the workflow for you, depending on the amount of work that needs to be done.

`run_single()`, local single thread

Runs your workflow in the same thread as the caller. This function is mainly for testing. When running workflows you almost always want to use one of the other functions.

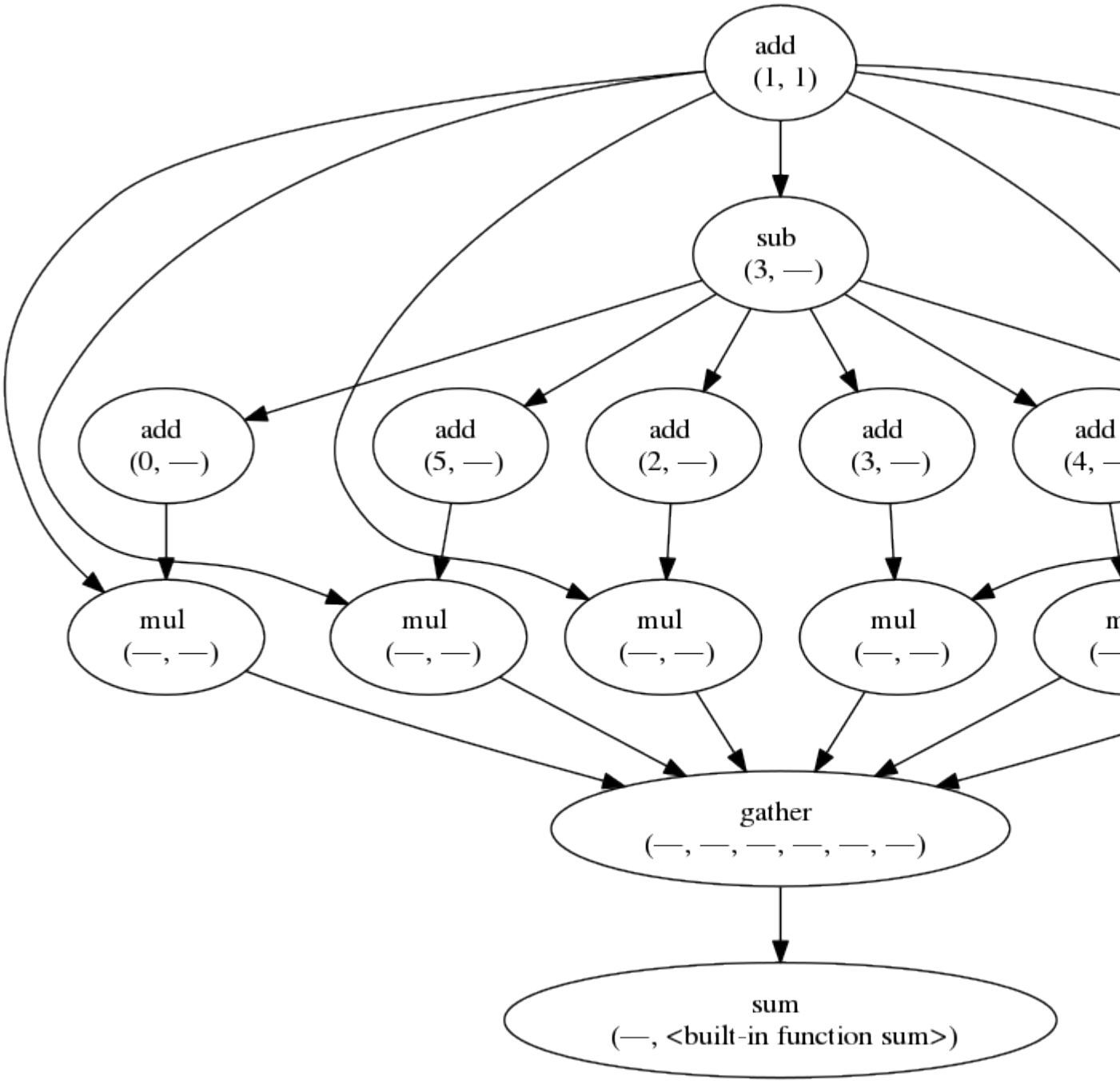


Fig. 2: The workflow graph of the second example.

`run_parallel()`, local multi-thread

Runs your workflow in parallel using any number of threads. Usually, specifying the number of cores in your CPU will give optimal performance for this runner.

Note: If you are very **very** certain that your workflow will never need to scale to cluster-computing, this runner is more lenient on the kinds of Python that is supported, because function arguments are not converted to and from JSON. Think of nested functions, lambda forms, generators, etc.

`run_process()`, local multi-process

Starts a second process to run jobs. This is usefull for testing the JSON compatability of your workflow on your own machine.

Xenon

Xenon is a Java library offering a uniform interface to all manners of job schedulers. Running a job on your local machine is as easy as submitting it to SLURM or Torque on your groceries supercomputer. To talk to Xenon from Python we use pyxenon.

Using the Xenon runner, there are two modes of operation: *batch* and *online*. In online mode, jobs are streamed to the worker and results read back. If your laptop crashes while an online computation is running, that is to say, the connection is broken, the worker dies and you may lose results. Getting the online mode to be more robust is one of the aims for upcomming releases.

The Xenon runner needs a way to setup the virtualenv on the remote side, so a worker script needs to be specified. We have included a bash-script `worker.sh` that should work in the simplest cases.

```
#!/bin/bash

# run in the directory where the script is located
cd "$(dirname "${BASH_SOURCE[0]}")"

# activate the virtualenv that is given as first argument
# invoking this script.
if [ -e $1/bin/activate ]; then
    source $1/bin/activate;
fi

# start the worker with the rest of the arguments.
# stderr is written to a file.
python -m noodles.worker ${@:2} 2> errlog

# close the virtualenv.
if [ -z ${VIRTUAL_ENV+x} ]; then
    deactivate;
fi
```

If you need to setup some more aspects of the environment, load modules, set variables etc., modify this script and put it in the directory where you want to run the jobs. Specify this directory in the Python script.

```
from noodles import (
    serial, gather)
```

(continues on next page)

(continued from previous page)

```
from noodles.run.xenon import (
    XenonConfig, RemoteJobConfig, XenonKeeper, run_xenon_prov)
from noodles.display import (
    NCDisplay)

from noodles.tutorial import add, accumulate

if __name__ == "__main__":
    a = [add(i, j) for i in range(5) for j in range(5)]
    b = accumulate(gather(*a))

    # XenonKeeper is the root Xenon object that gives access
    # to the Xenon Java library
    with XenonKeeper() as Xe:
        # We recommend loging in on your compute resource
        # through private/public key pairs. This prevents
        # passwords ending up as ASCII in your source files.
        certificate = Xe.credentials.newCertificateCredential(
            'ssh', os.environ['HOME'] + '/.ssh/id_rsa', '<username>', '', None)

        # Configure Xenon to access your favourite super computer.
        xenon_config = XenonConfig(
            jobs_scheme='slurm',
            location='login.super-duper-computer.darpa.net',
            credential=certificate
        )

        # Specify how to submit jobs.
        job_config = RemoteJobConfig(
            registry=serial.base,
            prefix='<path-to-virtualenv>',
            working_dir='<project-path>',
            time_out=5000
        )


        # Run jobs with NCurses based console feedback
        with NCDisplay() as display:
            result = run_xenon_prov(
                b, Xe, "cache.json", 2, xenon_config, job_config,
                display=display)

    print("This test is working {0}%!".format(result))
```

Hybrid mode

We may have a situation where a workflow consists of some very heavy *compute* jobs and a lot of smaller jobs that do some bookkeeping. If we were to schedule all the menial jobs to a SLURM queue we actually slow down the computation through the overhead of job submission. The Noodles cook may provide the schedule functions with hints on the type of job the function represents. Depending on these hints we may dispatch the job to a remote worker or keep it on the local machine.

We provide an example on how to use the hybrid worker in the source.

If you really need to, it is not too complicated to develop your own job runner based on some of these examples. Elsewhere in this documentation we elaborate on the architecture and interaction between runners and the scheduler,

see: *The Noodles Scheduler*.

4.2 Cooking of Noodles (library docs)

The cooking of good Noodles can be tricky. We try to make it as easy as possible, but to write good Noodles you need to settle in a *functional style* of programming. The functions you design cannot write to some global state, or modify its arguments and expect these modifications to persist throughout the program. This is not a restriction of Noodles itself, this is a fundamental principle that applies to all possible frameworks for parallel and distributed programming. So get used to it!

Every function call in Noodles (that is, calls to scheduled function) can be visualised as a node in a call graph. You should be able to draw this graph conceptually when designing the program. Luckily there is (almost) always a way to write down non-functional code in a functional way.

Note: Golden Rule: if you modify something, return it.

4.2.1 Call by value

Suppose we have the following program

```
from noodles import (schedule, run_single)

@schedule
def double(x):
    return x['value'] * 2

@schedule
def add(x, y):
    return x + y

a = {'value': 4}
b = double(a)
a['value'] = 5
c = double(a)
d = add(b, c)

print(run_single(d))
```

If this were undecorated Python, the answer would be 18. However, the computation of this answer depends on the time-dependency of the Python interpreter. In Python, dictionaries are passed by reference. The promised object `b` then contains a reference to the dictionary in `a`. If we then change the value in this dictionary, the call producing the value of `b` is retroactively changed to double the value 5 instead of 4.

If Noodles is to evaluate this program correctly it needs to `deepcopy()` every argument to a scheduled function. There is another way to have the same semantics produce a correct result. This is by making `a` a promised object in the first place. The third solution is to teach your user *functional programming*. Deep copying function arguments can result in a significant performance penalty on the side of the job scheduler. In most applications that we target this is not the bottle neck.

Since we aim for the maximum ease of use for the end-user, we chose to enable call-by-value by default.

4.2.2 Monads (sort of)

We still have ways to do object oriented programming and assignments. The `PromisedObject` class has several magic methods overloaded to translate to functional equivalents.

Member assignment

Especially member assignment is treated in a particular way. Suppose `a` is a `PromisedObject`, then the statement

```
a.b = 3
```

is (conceptually) transformed into

```
a = _setattr(a, 'b', 3)
```

where `_setattr()` is a scheduled function. The `PromisedObject` contains a representation of the complete workflow representing the computation to get to the value of `a`. In member assignment, this workflow is replaced with the new workflow containing this last instruction.

This is not a recommended way of programming. Every assignment results in a nested function call. The `statefulness` of the program is then implemented in the composition of functions, similar to how other functional languages do it using `monads`. It results in sequential code that will not parallelise so well.

Other magic methods

Next to member assignment, we also (obviously) support member reference, method function call and object function call (with `__call__`).

4.2.3 Storable

4.2.4 Serialisation

4.3 Tutorials

4.3.1 First Steps

This tutorial is also available in the form of a Jupyter Notebook. Try it out, and play!

Noodles is there to make your life easier, *in parallel*! The reason why Noodles can be easy and do parallel Python at the same time is its *functional* approach. In one part you'll define a set of functions that you'd like to run with Noodles, in an other part you'll compose these functions into a *workflow graph*. To make this approach work a function should not have any *side effects*. Let's not linger and just start noodling! First we define some functions to use.

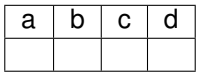
[1]:
```
from noodles import schedule

@schedule
def add(x, y):
    return x + y

@schedule
def mul(x,y):
    return x * y
```

Now we can create a workflow composing several calls to this function.

[2]:
```
a = add(1, 1)
b = mul(a, 2)
c = add(a, a)
d = mul(b, c)
```

That looks easy enough; the funny thing is though, that nothing has been computed yet! Noodles just created the workflow graphs corresponding to the values that still need to be computed. Until such time, we work with the *promise* of a future value. Using some function in `pygraphviz` we can look at the call graphs.

[3]:
```
from noodles.tutorial import display_workflows

display_workflows(prefix='first_steps-workflow',
                  a=a, b=b, c=c, d=d)
```



Now, to compute the result we have to tell Noodles to evaluate the program.

[4]:
```
from noodles import run_parallel

run_parallel(d, n_threads=2)
```

[4]:
```
16
```

4.3.2 Real World Tutorial 1: Translating Poetry

First example

We build workflows by calling functions. The simplest example of this is the "diamond workflow":

[1]:
```
from noodles import run_single
from noodles.tutorial import (add, sub, mul)

u = add(5, 4)
v = sub(u, 3)
w = sub(u, 2)
x = mul(v, w)

answer = run_single(x)

print("The answer is {0}.".format(answer))
```

```
The answer is 42.
```

That looks like any other Python code! But this example is a bit silly. How do we leverage Noodles to earn an honest living? Here's a slightly less silly example (but only just!). We will build a small translation engine that translates sentences by submitting each word to an online dictionary over a Rest API. To do this we make loops ("For thou shalt make loops of blue"). First we build the program as you would do in Python, then we sprinkle some Noodles magic and make it work parallel! Furthermore, we'll see how to:

- make more loops
- cache results for reuse

Making loops

Thats all swell, but how do we make a parallel loop? Let's look at a `map` operation; in Python there are several ways to perform a function on all elements in an array. For this example, we will translate some words using the Glosbe service, which has a nice REST interface. We first build some functionality to use this interface.

[2]:
```
import urllib.request
import json
import re


class Translate:
    """Translate words and sentences in the worst possible way. The Glosbe dictionary
    has a nice REST interface that we query for a phrase. We then take the first␣
↪result.
    To translate a sentence, we cut it in pieces, translate it and paste it back into
    a Frankenstein monster."""
    def __init__(self, src_lang='en', tgt_lang='fy'):
        self.src = src_lang
        self.tgt = tgt_lang
        self.url = 'https://glosbe.com/gapi/translate?' \
                   'from={src}&dest={tgt}&' \
                   'phrase={{phrase}}&format=json'.format(
                        src=src_lang, tgt=tgt_lang)

    def query_phrase(self, phrase):
        with urllib.request.urlopen(self.url.format(phrase=phrase.lower())) as␣
↪response:
            translation = json.loads(response.read().decode())
        return translation

    def word(self, phrase):
        translation = self.query_phrase(phrase)
        #translation = {'tuc': [{'phrase': {'text': phrase.lower()[::-1]}}]}
        if len(translation['tuc']) > 0 and 'phrase' in translation['tuc'][0]:
            result = translation['tuc'][0]['phrase']['text']
            if phrase[0].isupper():
                return result.title()
            else:
                return result
        else:
            return "<" + phrase + ">"

    def sentence(self, phrase):
        words = re.sub("[^\w]", " ", phrase).split()
        space = re.sub("[\w]+", "{}", phrase)
        return space.format(*map(self.word, words))
```

We start with a list of strings that desparately need translation. And add a little routine to print it in a gracious manner.

[3]:
```
shakespeare = [
    "If music be the food of love, play on,",
    "Give me excess of it; that surfeiting,",
    "The appetite may sicken, and so die."]

def print_poem(intro, poem):
    print(intro)
    for line in poem:
```

(continues on next page)

(continued from previous page)

```
        print("     ", line)
    print()

print_poem("Original:", shakespeare)
```

```
Original:
      If music be the food of love, play on,
      Give me excess of it; that surfeiting,
      The appetite may sicken, and so die.
```

Beginning Python programmers like to append things; this is not how you are supposed to program in Python; if you do, please go and read Jeff Knupp's *Writing Idiomatic Python*.

[4]:
```
shakespeare_auf_deutsch = []
for line in shakespeare:
    shakespeare_auf_deutsch.append(
        Translate('en', 'de').sentence(line))
print_poem("Auf Deutsch:", shakespeare_auf_deutsch)
```

```
Auf Deutsch:
      Wenn Musik sein der Essen von Minne, spielen an,
      Geben ich Übermaß von es; das übersättigend,
      Der Appetit dürfen erkranken, und so sterben.
```

Rather use a comprehension like so:

[5]:
```
shakespeare_ynt_frysk = \
    (Translate('en', 'fy').sentence(line) for line in shakespeare)
print_poem("Yn it Frysk:", shakespeare_ynt_frysk)
```

```
Yn it Frysk:
      At muzyk wêze de fiedsel fan leafde, boartsje oan,
      Jaan <me> by fersin fan it; dat <surfeiting>,
      De <appetite> maaie <sicken>, en dus deagean.
```

Or use `map`:

[6]:
```
shakespeare_pa_dansk = \
    map(Translate('en', 'da').sentence, shakespeare)
print_poem("På Dansk:", shakespeare_pa_dansk)
```

```
På Dansk:
      Hvis musik være de mad af kærlighed, spil på,
      Give mig udskejelser af det; som <surfeiting>,
      De appetit må <sicken>, og så dø.
```

Noodlify!

If your connection is a bit slow, you may find that the translations take a while to process. Wouldn't it be nice to do it in parallel? How much code would we have to change to get there in Noodles? Let's take the slow part of the program and add a `@schedule` decorator, and run! Sadly, it is not that simple. We can add `@schedule` to the `word` method. This means that it will return a promise.

- Rule: *Functions that take promises need to be scheduled functions, or refer to a scheduled function at some level.*

We could write

```
return schedule(space.format)(*(self.word(w) for w in words))
```

in the last line of the `sentence` method, but the string format method doesn't support wrapping. We rely on getting the signature of a function by calling `inspect.signature`. In some cases of build-in function this raises an exception. We may find a work around for these cases in future versions of Noodles. For the moment we'll have to define a little wrapper function.

[7]:
```
from noodles import schedule


@schedule
def format_string(s, *args, **kwargs):
    return s.format(*args, **kwargs)


import urllib.request
import json
import re


class Translate:
    """Translate words and sentences in the worst possible way. The Glosbe dictionary
    has a nice REST interface that we query for a phrase. We then take the first␣
↪result.
    To translate a sentence, we cut it in pieces, translate it and paste it back into
    a Frankenstein monster."""
    def __init__(self, src_lang='en', tgt_lang='fy'):
        self.src = src_lang
        self.tgt = tgt_lang
        self.url = 'https://glosbe.com/gapi/translate?' \
                   'from={src}&dest={tgt}&' \
                   'phrase={{phrase}}&format=json'.format(
                        src=src_lang, tgt=tgt_lang)

    def query_phrase(self, phrase):
        with urllib.request.urlopen(self.url.format(phrase=phrase.lower())) as␣
↪response:
            translation = json.loads(response.read().decode())
        return translation

    @schedule
    def word(self, phrase):
        #translation = {'tuc': [{'phrase': {'text': phrase.lower()[::-1]}}]}
        translation = self.query_phrase(phrase)

        if len(translation['tuc']) > 0 and 'phrase' in translation['tuc'][0]:
            result = translation['tuc'][0]['phrase']['text']
            if phrase[0].isupper():
                return result.title()
            else:
                return result
        else:
            return "<" + phrase + ">"
```

(continues on next page)

(continued from previous page)

```
    def sentence(self, phrase):
        words = re.sub("[^\w]", " ", phrase).split()
        space = re.sub("[\w]+", "{}", phrase)
        return format_string(space, *map(self.word, words))

    def __str__(self):
        return "[{} -> {}]".format(self.src, self.tgt)

    def __serialize__(self, pack):
        return pack({'src_lang': self.src,
                     'tgt_lang': self.tgt})

    @classmethod
    def __construct__(cls, msg):
        return cls(**msg)
```

Let's take stock of the mutations to the original. We've added a `@schedule` decorator to `word`, and changed a function call in `sentence`. Also we added the `__str__` method; this is only needed to plot the workflow graph. Let's run the new script.

[8]:
```
from noodles import gather, run_parallel
from noodles.tutorial import get_workflow_graph

shakespeare_en_esperanto = \
    map(Translate('en', 'eo').sentence, shakespeare)

wf = gather(*shakespeare_en_esperanto)
workflow_graph = get_workflow_graph(wf._workflow)
result = run_parallel(wf, n_threads=8)
print_poem("Shakespeare en Esperanto:", result)
```

```
Shakespeare en Esperanto:
      Se muziko esti la manĝaĵo de ami, ludi sur,
      Doni mi eksceso de ĝi; ke <surfeiting>,
      La apetito povi naŭzi, kaj tiel morti.
```

The last peculiar thing that you may notice, is the `gather` function. It collects the promises that `map` generates and creates a single new promise. The definition of `gather` is very simple:

```
@schedule
def gather(*lst):
    return lst
```

The workflow graph of the Esperanto translator script looks like this:

[9]:
```
workflow_graph.attr(size='10')
workflow_graph
```

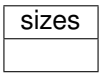
[9]:

Dealing with repetition

In the following example we have a line with some repetition.

[1]:
```
from noodles import (schedule, gather_all)
import re

@schedule
def word_size(word):
    return len(word)

@schedule
def format_string(s, *args, **kwargs):
    return s.format(*args, **kwargs)

def word_size_phrase(phrase):
    words = re.sub("[^\w]", " ", phrase).split()
    space = re.sub("[\w]+", "{}", phrase)
    word_lengths = map(word_size, words)
    return format_string(space, *word_lengths)
```

[2]:
```
from noodles.tutorial import display_workflows, run_and_print_log

display_workflows(
    prefix='poetry',
    sizes=word_size_phrase("Oote oote oote, Boe"))
```



Let's run the example workflows now, but focus on the actions taken, looking at the logs. The function `run_and_print_log` in the tutorial module runs our workflow with four parallel threads and caches results in a Sqlite3 database.

To see how this program is being run, we monitor the job submission, retrieval and result storage. First, should you have run this tutorial before, remove the database file.

[3]:
```
# remove the database if it already exists
!rm -f tutorial.db
```

Running the workflow, we can now see that at the second occurence of the word 'oote', the function call is attached to the first job that asked for the same result. The job `word_size('oote')` is run only once.

[4]:
```
run_and_print_log(word_size_phrase("Oote oote oote, Boe"), highlight=range(4, 8))
```

```
<IPython.core.display.HTML object>
```

[4]:
```
'4 4 4, 3'
```

Now, running a similar workflow again, notice that previous results are retrieved from the database.

[5]:
```
run_and_print_log(word_size_phrase("Oe oe oote oote oote"), highlight=range(5, 10))
```

```
<IPython.core.display.HTML object>
```

[5]:
```
'2 2 4 4 4'
```

Although the result of every single job is retrieved we still had to go through the trouble of looking up the results of `word_size('Oote')`, `word_size('oote')`, and `word_size('Boe')` to find out that we wanted the result from the `format_string`. If you want to cache the result of an entire workflow, pack the workflow in another scheduled function!

Versioning

We may add a version string to a function. This version is taken into account when looking up results in the database.

[6]:
```
@schedule(version='1.0')
def word_size_phrase(phrase):
    words = re.sub("[^\w]", " ", phrase).split()
    space = re.sub("[\w]+", "{}", phrase)
    word_lengths = map(word_size, words)
    return format_string(space, *word_lengths)

run_and_print_log(
    word_size_phrase("Kneu kneu kneu kneu ote kneu eur"),
    highlight=[1, 17])
```

```
<IPython.core.display.HTML object>
```

[6]:
```
'4 4 4 4 3 4 3'
```

See how the first job is evaluated to return a new workflow. Note that if the version is omitted, it is automatically generated from the source of the function. For example, let's say we decided the function `word_size_phrase` should return a dictionary of all word sizes in stead of a string. Here we use the function called `lift` to transform a dictionary containing promises to a promise of a dictionary. `lift` can handle lists, dictionaries, sets, tuples and objects that are constructable from their `__dict__` member.

[7]:
```
from noodles import lift

def word_size_phrase(phrase):
    words = re.sub("[^\w]", " ", phrase).split()
    return lift({word: word_size(word) for word in words})

display_workflows(prefix='poetry', lift=word_size_phrase("Kneu kneu kneu kneu ote␣
↪kneu eur"))
```

[8]:
```
run_and_print_log(word_size_phrase("Kneu kneu kneu kneu ote kneu eur"))
```

```
<IPython.core.display.HTML object>
```

[8]:
```
{'Kneu': 4, 'eur': 3, 'kneu': 4, 'ote': 3}
```

Be careful with versions! Noodles will believe you upon your word! If we lie about the version, it will go ahead and retrieve the result belonging to the old function:

[9]:
```
@schedule(version='1.0')
def word_size_phrase(phrase):
    words = re.sub("[^\w]", " ", phrase).split()
    return lift({word: word_size(word) for word in words})

run_and_print_log(
    word_size_phrase("Kneu kneu kneu kneu ote kneu eur"),
    highlight=[1])
```

```
<IPython.core.display.HTML object>
```

[9]:
```
'4 4 4 4 3 4 3'
```

4.3.3 Real World Tutorial 2: Boil, a make tool.

Boil: the source

```
#!/usr/bin/env python3

"""
Boil build utility.
"""

import argparse
import configparser
import subprocess
import sys
from itertools import chain
import os
import re


import noodles
from noodles.display import NCDisplay, DumbDisplay


def find_files(path, ext):
    """Find all files in `path` with extension `ext`. Returns an
    iterator giving tuples (folder, (files...)).

    :param path:
        search path
    :param ext:
        extension of files to find"""
    for folder, _, files in os.walk(path):
        for f in files:
            if f[-len(ext):] == ext:
                yield (folder, f)


def is_out_of_date(f, deps):
    """Check if file `f` needs to be updated. Returns True if any
    of the dependencies are newer than `f`.

    :param f:
        file to check
    :param deps:
        dependencies"""
    if not os.path.exists(f):
        return True

    f_stat = os.stat(f)

    for d in deps:
        d_stat = os.stat(d)

        if d_stat.st_mtime_ns > f_stat.st_mtime_ns:
            return True

    return False
```

(continues on next page)

(continued from previous page)

```
54
55
56  def dependencies(source_file, config):
57      """Find dependencies of source file.
58
59      :param source_file:
60          source file
61      :param config:
62          boil configuration"""
63      cmm = subprocess.run(
64          [config['cc'], '-MM', source_file] + config['cflags'].split(),
65          stdout=subprocess.PIPE, universal_newlines=True)
66      deps = re.sub('^.*: ', '', cmm.stdout) \
67          .replace('\\', '').replace('\n', '').split()
68
69      return deps
70
71
72  def object_filename(srcdir, filename, config):
73      """Create the object filename.
74
75      :param srcdir:
76          directory of source file
77      :param filename:
78          filename of source
79      :param config:
80          boil configuration"""
81      target_dir = os.path.join(config['objdir'], srcdir)
82
83      # if target directory doesn't exists, create it
84      # flag exists_ok=True for concurrency reasons
85      if not os.path.exists(target_dir):
86          os.makedirs(target_dir, exist_ok=True)
87
88      # construct name of object file
89      basename = os.path.splitext(filename)[0]
90      return os.path.join(target_dir, basename + '.o')
91
92
93  @noodles.schedule_hint(display="  Compiling {source_file} ... ",
94                         confirm=True)
95  @noodles.maybe
96  def compile_source(source_file, object_file, config):
97      """Compile a single source file."""
98      p = subprocess.run(
99          [config['cc'], '-c'] + config['cflags'].split() +
100         [source_file, '-o', object_file],
101         stderr=subprocess.PIPE, universal_newlines=True)
102     p.check_returncode()
103
104     return object_file
105
106
107 def get_object_file(src_dir, src_file, config):
108     """Ensures existence of up-to-date object file.
109
110     :param src_dir:
```

(continues on next page)

(continued from previous page)

```
        source directory
    :param src_file:
        source file
    :param config:
        boil configuration"""
    obj_path = object_filename(src_dir, src_file, config)
    src_path = os.path.join(src_dir, src_file)

    deps = dependencies(src_path, config)
    if is_out_of_date(obj_path, deps):
        return compile_source(src_path, obj_path, config)
    else:
        return obj_path


@noodles.schedule_hint(display="  Linking ... ",
                       confirm=True)
@noodles.maybe
def link(object_files, config):
    """Link object files to executable."""
    p = subprocess.run(
        [config['cc']] + object_files + ['-o', config['target']] +
        config['ldflags'].split(),
        stderr=subprocess.PIPE, universal_newlines=True)
    p.check_returncode()

    return config['target']


@noodles.schedule_hint(display="{msg}")
def message(msg, value=None):
    """Just print a message and pass on ``value``."""
    return value


@noodles.schedule
def get_target(obj_files, config):
    """Ensures that target is up-to-date.

    :param obj_files:
        list of object files
    :param config:
        boil configuration"""
    if any(noodles.failed(obj) for obj in obj_files):
        return Report(
            'failed',
            failures=[obj for obj in obj_files if noodles.failed(obj)])

    if is_out_of_date(config['target'], obj_files):
        return report_from_result(link(obj_files, config))
    else:
        return report_from_result('nothing-to-do')


@noodles.schedule
def report_from_result(result):
    """Assemble report from a result."""
```

(continues on next page)

(continued from previous page)

```
    if noodles.failed(result):
        return Report('failed', failures=[result])
    else:
        return Report('success', result=result)


class Report:
    """Contains status report of compile process."""
    def __init__(self, status, result=None, failures=None):
        self.status = status
        self.result = result
        self.failures = failures

    def __str__(self):
        line = '\033[31m' + '—' * 80 + '\033[m'

        def format_failure(failure):
            """Print a failure nicely."""
            return str(failure) + '\n' + line + '\n' + \
                failure.exception.stderr + line + '\n'

        if self.status == 'failed':
            return '\n\n'.join(map(format_failure, self.failures))
        else:
            return self.status


@noodles.schedule_hint(display="Building target {config[target]}")
def make_target(config):
    """Make a target. First compiles the source files, then
    links the object files to create an executable.

    :param config:
        boil configuration"""
    dirs = [config['srcdir']] + [
        os.path.normpath(os.path.join(config['srcdir'], d))
        for d in config['modules'].split()
    ]

    files = chain(
        *(find_files(d, config['ext'])
          for d in dirs))

    object_files = noodles.gather_all(
        get_object_file(src_dir, src_file, config)
        for src_dir, src_file in files)

    return get_target(object_files, config)


def read_config(filename):
    """Read configuration from `filename` and convert it to a nested dict.

    :param filename:
        name of an .ini file to read

    :returns:
```

(continues on next page)

(continued from previous page)

```
        dictionary."""
    reader = configparser.ConfigParser(
        interpolation=configparser.ExtendedInterpolation())
    reader.read(filename)

    return {k: dict(reader[k]) for k in reader.sections()}


def try_to_run(cmd, err_prefix):
    """Run a subprocess. Exit if subprocess fails."""
    process = subprocess.run(
        cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE,
        universal_newlines=True)
    try:
        process.check_returncode()
        return process.stdout
    except subprocess.CalledProcessError as exc:
        print(err_prefix, exc.stderr)
        sys.exit(1)


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description="Compile software. Configuration is in 'boil.ini'.")
    parser.add_argument(
        '-j', dest='n_threads', type=int, default=1,
        help='number of threads to run simultaneously.')
    parser.add_argument(
        '-dumb', default=False, action='store_true',
        help='print info without special term codes.')
    parser.add_argument(
        'target', type=str,
        help='target to build, give \'list\' to list targets.')
    args = parser.parse_args(sys.argv[1:])

    if not os.path.exists('boil.ini'):
        print("No boil.ini in current directory.")
        sys.exit(1)

    bconfig = read_config('boil.ini')

    if 'generic' not in bconfig:
        print("Error: Configuration has no 'generic' section.")
        sys.exit(1)

    if args.target == 'list':
        targets = list(bconfig.keys())
        targets.remove('generic')
        print("Possible targets: ", ', '.join(targets))
        sys.exit(0)

    if 'command' in bconfig[args.target]:
        os.system(bconfig[args.target]['command'])

    else:
        target_config = bconfig['generic']
        target_config.update(bconfig[args.target])
```

(continues on next page)

(continued from previous page)

```
282
283         if 'libraries' in target_config:
284             a = try_to_run(
285                 ['pkg-config', '--libs'] + target_config['libraries'].split(),
286                 err_prefix="Error running pkg-config: ")
287
288             target_config['ldflags'] += ' ' + a
289
290             a = try_to_run(
291                 ['pkg-config', '--cflags'] +
292                 target_config['libraries'].split(),
293                 err_prefix="Error running pkg-config: ")
294
295             target_config['cflags'] += ' ' + a
296
297         work = make_target(target_config)
298         display_type = DumbDisplay if args.dumb else NCDisplay
299         with display_type() as display:
300             report = noodles.run_logging(work, args.n_threads, display)
301         print(report)
```

This tutorial should teach you the basics of using Noodles to parallelise a Python program. We will see some of the quirks that come with programming in a strict functional environment.

"make -j"

The first of all workflow engines is called 'make'. It builds a tree of interdepending jobs and executes them in order. The '-j' option allows the user to specify a number of jobs to be run simultaneously to speed up execution. The syntax of make is notoriously terse, partly due to it's long heritage (from 1976). This example shows how we can write a script that compiles a C/C++ program using GCC or CLANG, then how we can parallelise it using Noodles. We have even included a small C++ program to play with!

Functions

Compiling a C program is a two-stage process. First we need to compile each source file to an object file, then link these object files to an executable. The `compile_source()` function looks like this:

```
1 @schedule
2 def compile_source(source_file, object_file, config):
3     p = subprocess.run(
4         [config['cc'], '-c'] + config['cflags'].split() \
5             + [source_file, '-o', object_file],
6         stderr=subprocess.PIPE, universal_newlines=True)
7     p.check_returncode()
8
9     return object_file
```

It takes a source file, an object file and a configuration object. This configuration contains all the information on which compiler to use, with which flags, and so on. If the compilation was successful, the name of the object file is returned. This last bit is crucial if we want to include this function in a workflow.

Note: Each dependency (in this case linking to an excecutable requires compilation first) should be represented by return values of one function ending up as arguments to another function.

The function for linking object files to an executable looks very similar:

```
@schedule
def link(object_files, config):
    p = subprocess.run(
        [config['cc']] + object_files + ['-o', config['target']] \
            + config['ldflags'].split(),
        stderr=subprocess.PIPE, universal_newlines=True)
    p.check_returncode()

    return config['target']
```

In this case the function takes a list of object file names and the same configuration object that we saw before. Again, this function returns the name of the target executable file. The caller of this function already knows the name of the target file, but we need it to track dependencies between function calls.

Since both the `link()` and the `compile_source()` functions do actual work that we'd like to see being done in a concurrent environment, they need to be decorated with the `schedule()` decorator.

One of the great perks of using Make, is that it skips building any files that are already up-to-date with the source code. If our little build script is to compete with such efficiency, we should do the same!

```
def is_out_of_date(f, deps):
    if not os.path.exists(f):
        return True

    f_stat = os.stat(f)

    for d in deps:
        d_stat = os.stat(d)

        if d_stat.st_mtime_ns > f_stat.st_mtime_ns:
            return True

    return False
```

This function takes a file `f` and a list of files `deps` and checks the modification dates of all of the files in `deps` against that of `f`.

One of the *quirks*, that we will need to deal with, is that some *logic* in a program needs to have knowledge of the actual objects that are computed, not just the possibility of such an object in the future. When designing programs for Noodles, you need to be aware that such logic can only be performed *inside* the functions. Say we have a condition under which one or the other action needs to be taken, and this condition depends on the outcome of a previous element in the workflow. The actual Python `if` statement evaluating this condition needs to be inside a scheduled function. One way around this, is to write a wrapper:

```
@schedule
def cond(predicate, when_true, when_false):
    if predicate:
        return when_true
    else:
        return when_false
```

However, there is a big problem with this approach! The Noodles engine sees two arguments to the `cond` function that it wants to evaluate before heading into the call to `cond`. Both arguments will be evaluated before we can even decide which of the two we should use! We will present a solution to this problem at a later stage, but for now we will have to work our way around this by embedding the conditional logic in a more specific function.

In this case we have the function `is_out_of_date` that determines whether we need to recompile a file or leave

it as it is. The second stage, linking the object files to an executable, only needs to happen if any of the object files is younger than the executable. However these object files are part of the logic inside the workflow! The conditional execution of the linker needs to be called by another scheduled function:

```
@schedule
def get_target(obj_files, config):
    if is_out_of_date(config['target'], obj_files):
        return link(obj_files, config)
    else:
        return message("target is up-to-date.")
```

Since we need the answer to `is_out_of_date()` in the present, the `is_out_of_date()` function cannot be a scheduled function. Python doesn't know the truth value of a `PromisedObject`. The `message` function is not a special function; it just prints a message and returns a value (optional second argument). We still need to optionalise the compilation step. Since all of the information needed to decide whether to compile or not is already present, we can make this a normal Python function; there is no need to schedule anything (even though everything would still work if we did):

```
def get_object_file(src_dir, src_file, config):
    obj_path = object_filename(src_dir, src_file, config)
    src_path = os.path.join(src_dir, src_file)

    deps = dependencies(src_path, config)
    if is_out_of_date(obj_path, deps):
        return compile_source(src_path, obj_path, config)
    else:
        return obj_path
```

The `object_filename` is a little helper function creating a sensible name for the object file; also it makes sure that the directory in which the object file is placed exists. `dependencies` Runs the compiler with '-MM' flags to obtain the header dependencies of the C-file.

We are now ready to put these functions together in a workflow!

```
def make_target(config):
    dirs = [config['srcdir']] + [
        os.path.normpath(os.path.join(config['srcdir'], d))
        for d in config['modules'].split()
    ]

    files = chain(*(
        find_files(d, config['ext'])
        for d in dirs)
    )

    obj_files = noodles.gather(*[
        get_object_file(src_dir, src_file, config)
        for src_dir, src_file in files
        ])

    return get_target(obj_files, config)
```

Let's go through this step-by-step. The `make_target` function takes one argument, the config object. We obtain from the configuration the directories to search for source files. We then search these directories for any files with the correct file extension, stored in `config['ext']`. The variable `files` now contains a list of pairs, each pair having a directory and file name. So far we have not yet used any Noodles code.

Next we pass each source file through the `get_object_file` function in a list comprehension. The resulting list

contains both `PromisedObject` and string values; strings for all the object files that are already up-to-date. To pass this list to the linking stage we have to make sure that Noodles understands that the list is something that is being promised. If we were to pass it as is, Noodles just sees a list as an argument to `get_target` and doesn't look any deeper.

Note: Every `PromisedObject` has to be passed as an argument to a scheduled function in order to be evaluated. To pass a list to a scheduled function, we have to convert the list of promises into a promise of a list.

The function `gather()` solves this little problem; it's definition is very simple:

```
@schedule
def gather(*args):
    return list(args)
```

Now that the variable `obj_files` is a `PromisedObject`, we can pass it to `get_target`, giving us the final workflow. Running this workflow can be as simple as `run_single(wf)` or `run_parallel(wf, n_threads=4)`.

Friendly output and error handling

The code as defined above will run, but if the compiler gives error messages it will crash in a very ugly manner. Noodles has some features that will make our fledgeling Make utility much prettier. We can decorate our functions further with information on how to notify the user of things happening:

```
@schedule_hint(display="Compiling {source_file} ... ",
               confirm=True)
def compile_source(source_file, object_file, config):
    pass
```

The `schedule_hint()` decorator attaches hints to the scheduled function. These hints can be used in any fashion we like, depending on the workers that we use to evaluate the workflow. In this case we use the `run_logging()` worker, with the `SimpleDisplay` class to take care of screen output:

```
with SimpleDisplay(error_filter) as display:
    noodles.run_logging(work, args.n_threads, display)
```

The `error_filter` function determines what errors are expected and how we print the exceptions that are caught. In our case we expect exceptions of type `subprocess.CalledProcessError` in the case of a compiler error. Otherwise the exception is unexpected and should be treated as a bug in Boil!

```
1 def error_filter(xcptn):
2     if isinstance(xcptn, subprocess.CalledProcessError):
3         return xcptn.stderr
4     else:
5         return None
```

The `display` tag in the hinst tells the `display` object to print a text when the job is started. The `confirm` flag in the hints tells the `display` object that a function is error-prone and to draw a green checkmark on success and a red X in case of failure.

Conclusion

You should now be able to fully understand the sourcecode of Boil! Try it out on the sample code provided:

```
> ./boil --help
```

```
usage: boil [-h] [-j N_THREADS] [-dumb] target

Compile software. Configuration is in 'boil.ini'.

positional arguments:
  target        target to build, give 'list' to list targets.

optional arguments:
  -h, --help    show this help message and exit
  -j N_THREADS  number of threads to run simultaneously.
  -dumb         print info without special term codes.
```

```
> cat boil.ini
```

```
[generic]
objdir = obj
ldflags = -lm
cflags = -g -std=c++11 -O2 -fdiagnostics-color -Wpedantic
cc = g++
ext = .cc

[main]
srcdir = main
target = hello
modules = ../src

[test]
srcdir = test
target = unittests
modules = ../src

[clean]
command = rm -rf ${generic:objdir} ${test:target} ${main:target}
```

```
> ./boil main -j4
```

```
─(Building target hello)
│   Compiling src/mandel.cc ...                    (✓)
│   Compiling src/common.cc ...                    (✓)
│   Compiling src/render.cc ...                    (✓)
│   Compiling src/iterate.cc ...                   (✓)
│   Compiling src/julia.cc ...                     (✓)
│   Compiling main/main.cc ...                     (✓)
│   Linking ...                                    (✓)
─(success)
```

4.3.4 Real World Tutorial 3: Parallel Number crunching using Cython

Python was not designed to be very good at parallel processing. There are two major problems at the core of the language that make it hard to implement parallel algorithms.

- The Global Interpreter Lock

- Flexible object model

The first of these issues is the most famous obstacle towards a convincing multi-threading approach, where a single instance of the Python interpreter runs in several threads. The second point is more subtle, but makes it harder to do multi-processing, where several independent instances of the Python interpreter work together to achieve parallelism. We will first explain an elegant way to work around the Global Interpreter Lock, or GIL: use Cython.

Using Cython to lift the GIL

The GIL means that the Python interpreter will only operate on one thread at a time. Even when we think we run in a gazillion threads, Python itself uses only one. Multi-threading in Python is only usefull to wait for I/O and to perform system calls. To do useful CPU intensive work in multi-threaded mode, we need to develop functions that are implemented in C, and tell Python when we call these functions not to worry about the GIL. The easiest way to achieve this, is to use Cython. We develop a number-crunching prime adder, and have it run in parallel threads.

We'll load the `multiprocessing`, `threading` and `queue` modules to do our plumbing, and the `cython` extension so we can do the number crunching, as is shown in this blog post.

[1]:
```
%load_ext cython
import multiprocessing
import threading
import queue
```

We define a function that computes the sum of all primes below a certain integer `n`, and don't try to be smart about it; the point is that it needs a lot of computation. These functions are designated `nogil`, so that we can be certain no Python objects are accessed. Finally we create a single Python exposed function that uses the:

```
with nogil:
    ...
```

statement. This is a context-manager that lifts the GIL for the duration of its contents.

[2]:
```
%%cython

from libc.math cimport ceil, sqrt


cdef inline int _is_prime(int n) nogil:
    """return a boolean, is the input integer a prime?"""
    if n == 2:
        return True
    cdef int max_i = <int>ceil(sqrt(n))
    cdef int i = 2
    while i <= max_i:
        if n % i == 0:
            return False
        i += 1
    return True


cdef unsigned long _sum_primes(int n) nogil:
    """return sum of all primes less than n """
    cdef unsigned long i = 0
    cdef int x
    for x in range(2, n):
        if _is_prime(x):
            i += x
```

(continues on next page)

(continued from previous page)

```
    return i


def sum_primes(int n):
    with nogil:
        result = _sum_primes(n)
    return result
```

In fact, we only loaded the `multiprocessing` module to get the number of CPUs on this machine. We also get a decent amount of work to do in the `input_range`.

[3]:
```
input_range = range(int(1e6), int(2e6), int(5e4))
ncpus = multiprocessing.cpu_count()
print("We have {} cores to work on!".format(ncpus))
```

```
We have 4 cores to work on!
```

Single thread

Let's first run our tests in a single thread:

[4]:
```
%%time

for i in input_range:
    print(sum_primes(i), end=' ', flush=True)
print()
```

```
37550402023 41276629127 45125753695 49161463647 53433406131 57759511224 62287995772␣
↪66955471633 71881256647 76875349479 82074443256 87423357964 92878592188 98576757977␣
↪104450958704 110431974857 116581137847 122913801665 129451433482 136136977177
CPU times: user 8.62 s, sys: 5.05 ms, total: 8.62 s
Wall time: 8.63 s
```

Multi-threading: Worker pool

We can do better than that! We now create a queue containing the work to be done, and a pool of threads eating from this queue. The workers will keep on working as long as the queue has work for them.

[5]:
```
%%time

### We need to define a worker function that fetches jobs from the queue.
def worker(q):
    while True:
        try:
            x = q.get(block=False)
            print(sum_primes(x), end=' ', flush=True)
        except queue.Empty:
            break

### Create the queue, and fill it with input values
work_queue = queue.Queue()
for i in input_range:
    work_queue.put(i)
```

(continues on next page)

(continued from previous page)

```
### Start a number of threads
threads = [
    threading.Thread(target=worker, args=(work_queue,))
    for i in range(ncpus)]

for t in threads:
    t.start()

### Wait until all of them are done
for t in threads:
    t.join()

print()
```

```
37550402023 41276629127 45125753695 49161463647 53433406131 57759511224 62287995772␣
↪66955471633 71881256647 76875349479 82074443256 87423357964 92878592188 98576757977␣
↪104450958704 110431974857 116581137847 122913801665 129451433482 136136977177
CPU times: user 14.7 s, sys: 9.1 ms, total: 14.7 s
Wall time: 3.98 s
```

Using Noodles

On my laptop, a dual-core hyper-threaded `Intel(R) Core(TM) i5-5300U CPU`, this runs just over two times faster than the single threaded code. However, setting up a queue and a pool of workers is quite cumbersome. Also, this approach doesn't scale up if the dependencies between our computations get more complex. Next we'll use Noodles to provide the multi-threaded environment to execute our work. We'll need three functions:

- `schedule` to decorate our work function
- `run_parallel` to run the work in parallel
- `gather` to collect our work into a workflow

[6]:
```
from noodles import (schedule, run_parallel, gather)
```

[7]:
```
%%time

@schedule
def s_sum_primes(n):
    result = sum_primes(n)
    print(result, end=' ', flush=True)
    return result

p_prime_sums = gather(*(s_sum_primes(i) for i in input_range))
prime_sums = run_parallel(p_prime_sums, n_threads=ncpus)
print()
```

```
37550402023 41276629127 45125753695 49161463647 53433406131 57759511224 62287995772␣
↪66955471633 71881256647 76875349479 82074443256 87423357964 92878592188 98576757977␣
↪104450958704 110431974857 116581137847 122913801665 129451433482 136136977177
CPU times: user 14.7 s, sys: 11.8 ms, total: 14.7 s
Wall time: 4.08 s
```

That does look much nicer! We have much less code, the code we do have is clearly separating function and form, and this approach is easily expandable to more complex situations.

4.3.5 Expecting the unexpected

To handle errors properly deserves a chapter on its own in any programming book. Python gives us many ways do deal with errors fatal and otherwise: try, except, assert, if ... Using these mechanisms in a naive way may lead to code that is littered with safety `if` statements and `try-except` blocks, just because we need to account for errors at every level in a program.

In this tutorial we'll see how we can use exceptions in a more effective way. As an added bonus we learn how to use exceptions in a manner that is compatible with the Noodles programming model. Let's try something dangerous! We'll compute the reciprocal of a list of numbers. To see what is happening, the function `something_dangerous` contains a print statement.

[1]:
```
import sys

def something_dangerous(x):
    print("computing reciprocal of", x)
    return 1 / x

try:
    for x in [2, 1, 0, -1]:
        print("1/{} = {}".format(x, something_dangerous(x)))

except ArithmeticError as error:
    print("Something went terribly wrong:", error)
```

```
computing reciprocal of 2
1/2 = 0.5
computing reciprocal of 1
1/1 = 1.0
computing reciprocal of 0
Something went terribly wrong: division by zero
```

This shows how exceptions are raised and caught, but this approach is somewhat limited. Suppose now, that we weren't expecting this expected unexpected behaviour and we wanted to compute everything before displaying our results.

[2]:
```
input_list = [2, 1, 0, -1]
reciprocals = [something_dangerous(item)
               for item in input_list]

print("The reciprocal of", input_list, "is", reciprocals)
```

```
computing reciprocal of 2
computing reciprocal of 1
computing reciprocal of 0
```

```
---------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-5d396078122a> in <module>()
      1 input_list = [2, 1, 0, -1]
      2 reciprocals = [something_dangerous(item)
---> 3                for item in input_list]
      4
      5 print("The reciprocal of", input_list, "is", reciprocals)

<ipython-input-2-5d396078122a> in <listcomp>(.0)
      1 input_list = [2, 1, 0, -1]
      2 reciprocals = [something_dangerous(item)
---> 3                for item in input_list]
      4
```

(continues on next page)

(continued from previous page)

```
      5 print("The reciprocal of", input_list, "is", reciprocals)

<ipython-input-1-990ff89c780e> in something_dangerous(x)
      3 def something_dangerous(x):
      4     print("computing reciprocal of", x)
---> 5     return 1 / x
      6
      7 try:

ZeroDivisionError: division by zero
```

Ooops! Let's fix that.

[3]:
```
try:
    reciprocals = [something_dangerous(item)
                   for item in input_list]

except ArithmeticError as error:
    print("Something went terribly wrong:", error)

else:
    print("The reciprocal of\n\t", input_list,
          "\nis\n\t", reciprocals)
```

```
computing reciprocal of 2
computing reciprocal of 1
computing reciprocal of 0
Something went terribly wrong: division by zero
```

That's also not what we want. We wasted all this time computing nice reciprocals of numbers, only to find all of our results being thrown away because of one stupid zero in the input list. We can fix this.

[4]:
```
import math

def something_safe(x):
    try:
        return something_dangerous(x)
    except ArithmeticError as error:
        return math.nan

reciprocals = [something_safe(item)
               for item in input_list]

print("The reciprocal of\n\t", input_list,
      "\nis\n\t", reciprocals)
```

```
computing reciprocal of 2
computing reciprocal of 1
computing reciprocal of 0
computing reciprocal of -1
The reciprocal of
         [2, 1, 0, -1]
is
         [0.5, 1.0, nan, -1.0]
```

That's better! We skipped right over the error and continued to more interesting results. So how are we going to make this solution more generic? Subsequent functions may not know how to handle that little `nan` in our list.

[5]:
```
square_roots = [math.sqrt(item) for item in reciprocals]
```

```
---------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-4d8b46ef9954> in <module>()
---> 1 square_roots = [math.sqrt(item) for item in reciprocals]

<ipython-input-5-4d8b46ef9954> in <listcomp>(.0)
---> 1 square_roots = [math.sqrt(item) for item in reciprocals]

ValueError: math domain error
```

Hmmmpf. There we go again.

[6]:
```
def safe_sqrt(x):
    try:
        return math.sqrt(x)
    except ValueError as error:
        return math.nan

[safe_sqrt(item) for item in reciprocals]
```

[6]:
```
[0.7071067811865476, 1.0, nan, nan]
```

This seems Ok, but there are two problems here. For one, it feels like we're doing too much work! We have a repeating code pattern here. That's always a moment to go back and consider making parts of our code more generic. At the same time, this is when we need some more advanced Python concepts to get us out of trouble. We're going to define a function in a function!

[7]:
```
def secure_function(dangerous_function):
    def something_safe(x):
        """A safer version of something dangerous."""
        try:
            return dangerous_function(x)
        except (ArithmeticError, ValueError):
            return math.nan

    return something_safe
```

Consider what happens here. The function `secure_function` takes a function `something_dangerous` as an argument and returns a new function `something_safe`. This new function executes `something_dangerous` within a `try-except` block to deal with the possibility of failure. Let's see how this works.

[8]:
```
safe_sqrt = secure_function(math.sqrt)
print("2 =", safe_sqrt(2))
print("-1 =", safe_sqrt(-1))
print()
help(safe_sqrt)
```

```
2 = 1.4142135623730951
-1 = nan

Help on function something_safe in module __main__:

something_safe(x)
    A safer version of something dangerous.
```

Ok, so that works! However, the documentation of `safe_sqrt` is not yet very useful. There is a nice library routine

that may help us here: `functools.wraps`; this utility function sets the correct name and doc-string to our new function.

[9]:
```
import functools

def secure_function(dangerous_function):
    """Create a function that doesn't raise ValueErrors."""
    @functools.wraps(dangerous_function)
    def something_safe(x):
        """A safer version of something dangerous."""
        try:
            return dangerous_function(x)
        except (ArithmeticError, ValueError):
            return math.nan

    return something_safe
```

[10]:
```
safe_sqrt = secure_function(math.sqrt)
help(safe_sqrt)
```

```
Help on function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

Now it is very easy to also rewrite our function computing the reciprocals safely:

[11]:
```
something_safe = secure_function(something_dangerous)
[safe_sqrt(something_safe(item)) for item in input_list]
```

```
computing reciprocal of 2
computing reciprocal of 1
computing reciprocal of 0
computing reciprocal of -1
```

[11]:
```
[0.7071067811865476, 1.0, nan, nan]
```

There is a second problem to this approach, which is a bit more subtle. How do we know where the error occured? We got two values of `nan` and are desperate to find out what went wrong. We'll need a little class to capture all aspects of failure.

[12]:
```
class Fail:
    """Keep track of failures."""
    def __init__(self, exception, trace):
        self.exception = exception
        self.trace = trace

    def extend_trace(self, f):
        """Grow a stack trace."""
        self.trace.append(f)
        return self

    def __str__(self):
        return "Fail in " + " -> ".join(
            f.__name__ for f in reversed(self.trace)) \
```

(continues on next page)

(continued from previous page)

```
            + ":\n\t" + type(self.exception).__name__ \
            + ": " + str(self.exception)
```

We will adapt our earlier design for `secure_function`. If the given argument is a `Fail`, we don't even attempt to run the next function. In stead, we extend the trace of the failure, so that we can see what happened later on.

[13]:
```
def secure_function(dangerous_function):
    """Create a function that doesn't raise ValueErrors."""
    @functools.wraps(dangerous_function)
    def something_safe(x):
        """A safer version of something dangerous."""
        if isinstance(x, Fail):
            return x.extend_trace(dangerous_function)
        try:
            return dangerous_function(x)
        except Exception as error:
            return Fail(error, [dangerous_function])

    return something_safe
```

Now we can rewrite our little program entirely from scratch:

[14]:
```
@secure_function
def reciprocal(x):
    return 1 / x

@secure_function
def square_root(x):
    return math.sqrt(x)

reciprocals = map(reciprocal, input_list)
square_roots = map(square_root, reciprocals)

for x, result in zip(input_list, square_roots):
    print("sqrt( 1 /", x, ") =", result)
```

```
sqrt( 1 / 2 ) = 0.7071067811865476
sqrt( 1 / 1 ) = 1.0
sqrt( 1 / 0 ) = Fail in square_root -> reciprocal:
        ZeroDivisionError: division by zero
sqrt( 1 / -1 ) = Fail in square_root:
        ValueError: math domain error
```

See how we retain a trace of the functions that were involved in creating the failed state, even though the execution of that produced those values is entirely decoupled. This is exactly what we need to trace errors in Noodles.

Handling errors in Noodles

Noodles has the functionality of `secure_function` build in by the name of `maybe`. The following code implements the above example in terms of `noodles.maybe`:

[1]:
```
import noodles
import math
from noodles.tutorial import display_workflows

@noodles.maybe
```

(continues on next page)

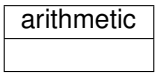
(continued from previous page)

```
def reciprocal(x):
    return 1 / x

@noodles.maybe
def square_root(x):
    return math.sqrt(x)

results = [square_root(reciprocal(x)) for x in [2, 1, 0, -1]]
for result in results:
    print(str(result))
```

```
0.7071067811865476
1.0
Fail: __main__.square_root (<ipython-input-1-74755080cfd2>:9)
* failed arguments:
    __main__.square_root `0` Fail: __main__.reciprocal␣
↪(<ipython-input-1-74755080cfd2>:5)
    * ZeroDivisionError: division by zero
Fail: __main__.square_root (<ipython-input-1-74755080cfd2>:9)
* ValueError: math domain error
```

The `maybe` decorator works well together with `schedule`. The following workflow is full of errors!

[16]:
```
@noodles.schedule
@noodles.maybe
def add(a, b):
    return a + b

workflow = add(noodles.schedule(reciprocal)(0),
               noodles.schedule(square_root)(-1))
display_workflows(arithmetic=workflow, prefix='errors')
```



Both the reciprocal and the square root functions will fail. Noodles is smart enough to report on both errors.‘

[17]:
```
result = noodles.run_single(workflow)
print(result)
```

```
Fail: __main__.add (<ipython-input-16-ca83c3781f78>:1)
* failed arguments:
    __main__.add `0` Fail: __main__.reciprocal (<ipython-input-15-74755080cfd2>:5)
    * ZeroDivisionError: division by zero
    __main__.add `1` Fail: __main__.square_root (<ipython-input-15-74755080cfd2>:9)
    * ValueError: math domain error
```

Example: parallel stat

Let’s do an example that works with external processes. The UNIX command `stat` gives the status of a file

[18]:
```
!stat -t -c '%A %10s %n' /dev/null
```

```
crw-rw-rw-          0 /dev/null
```

If a file does note exist, `stat` returns an error-code of 1.

[19]:
```
!stat -t -c '%A %10s %n' does-not-exist
```

```
stat: kan status van 'does-not-exist' niet opvragen: No such file or directory
```

We can wrap the execution of the `stat` command in a helper function.

[20]:
```
from subprocess import run, PIPE, CalledProcessError

@noodles.schedule
@noodles.maybe
def stat_file(filename):
    p = run(['stat', '-t', '-c', '%A %10s %n', filename],
            check=True, stdout=PIPE, stderr=PIPE)
    return p.stdout.decode().strip()
```

The `run` function runs the given command and returns a `CompletedProcess` object. The `check=True` argument enables checking for return value of the child process. If the return value is any other then 0, a `CalledProcessError` is raised. Because we decorated our function with `noodles.maybe`, such an error will be caught and a `Fail` object will be returned.

[21]:
```

files = ['/dev/null', 'does-not-exist', '/home', '/usr/bin/python3']
workflow = noodles.gather_all(stat_file(f) for f in files)
display_workflows(stat=workflow, prefix='errors')
```

We can now run this workflow and print the output in a table.

[22]:
```
result = noodles.run_parallel(workflow, n_threads=4)

for file, stat in zip(files, result):
    print('stat {:18} -> {}'.format(
        file, stat if not noodles.failed(stat)
        else 'failed: ' + stat.exception.stderr.decode().strip()))
```

```
stat /dev/null          -> crw-rw-rw-          0 /dev/null
stat does-not-exist     -> failed: stat: kan status van 'does-not-exist' niet␣
↪opvragen: No such file or directory
stat /home              -> drwxr-xr-x       4096 /home
stat /usr/bin/python3   -> lrwxrwxrwx          9 /usr/bin/python3
```

4.3.6 Serialising the Stars

Noodles lets you run jobs remotely and store/retrieve results in case of duplicate jobs or reruns. These features rely on the *serialisation* (and not unimportant, reconstruction) of all objects that are passed between scheduled functions. Serialisation refers to the process of turning any object into a stream of bytes from which we can reconstruct a functionally identical object. "Easy enough!" you might think, just use `pickle`.

[1]:
```
from noodles.tutorial import display_text
import pickle

function = pickle.dumps(str.upper)
```

(continues on next page)

(continued from previous page)

```
message = pickle.dumps("Hello, Wold!")

display_text("function: " + str(function))
display_text("message: " + str(message))
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

[2]:
```
pickle.loads(function)(pickle.loads(message))
```

[2]:
```
'HELLO, WOLD!'
```

However `pickle` cannot serialise all objects ... "Use `dill`!" you say; still the pickle/dill method of serializing is rather indiscriminate. Some of our objects may contain runtime data we can't or don't want to store, coroutines, threads, locks, open files, you name it. We work with a Sqlite3 database to store our data. An application might store gigabytes of numerical data. We don't want those binary blobs in our database, rather to store them externally in a HDF5 file.

There are many cases where a more fine-grained control of serialisation is in order. The bottom line being, that there is *no silver bullet solution*. Here we show some examples on how to customize the Noodles serialisation mechanism.

The registry

Noodles keeps a registry of `Serialiser` objects that know exactly how to serialise and reconstruct objects. This registry is specified to the backend when we call the one of the `run` functions. To make the serialisation registry visible to remote parties it is important that the registry can be imported. This is why it has to be a function of zero arguments (a *thunk*) returning the actual registry object.

```
def registry():
    return Registry(...)

run(workflow,
    db_file='project-cache.db',
    registry=registry)
```

The registry that should always be included is `noodles.serial.base`. This registry knows how to serialise basic Python dictionaries, lists, tuples, sets, strings, bytes, slices and all objects that are internal to Noodles. Special care is taken with objects that have a `__name__` attached and can be imported using the `__module__.__name__` combination.

Registries can be composed using the + operator. For instance, suppose we want to use `pickle` as a default option for objects that are not in `noodles.serial.base`:

[3]:
```
import noodles

def registry():
    return noodles.serial.pickle() \
        + noodles.serial.base()

reg = registry()
```

Let's see what is made of our objects!

[4]:
```
display_text(reg.to_json([
    "These data are JSON compatible!", 0, 1.3, None,
    {"dictionaries": "too!"}], indent=2))
```

```
<IPython.core.display.HTML object>
```

Great! JSON compatible data stays the same. Now try an object that JSON doesn't know about.

[5]:
```
display_text(reg.to_json({1, 2, 3}, indent=2), [1])
```

```
<IPython.core.display.HTML object>
```

Objects are encoded as a dictionary containing a `'_noodles'` key. So what will happen if we serialise an object the registry cannot possibly know about? Next we define a little astronomical class describing a star in the Morgan-Keenan classification scheme.

[6]:
```
class Star(object):
    """Morgan-Keenan stellar classification."""
    def __init__(self, spectral_type, number, luminocity_class):
        assert spectral_type in "OBAFGKM"
        assert number in range(10)

        self.spectral_type = spectral_type
        self.number = number
        self.luminocity_class = luminocity_class

rigel = Star('B', 8, 'Ia')
display_text(reg.to_json(rigel, indent=2), [4], max_width=60)
```

```
<IPython.core.display.HTML object>
```

The registry obviously doesn't know about `Stars`, so it falls back to serialisation using `pickle`. The pickled data is further encoded using `base64`. This solution won't work if some of your data cannot be pickled. Also, if you're sensitive to aesthetics, the pickled output doesn't look very nice.

serialize and *construct*

One way to take control of the serialisation of your objects is to add the `__serialize__` and `__construct__` methods.

[7]:
```
class Star(object):
    """Morgan-Keenan stellar classification."""
    def __init__(self, spectral_type, number, luminocity_class):
        assert spectral_type in "OBAFGKM"
        assert number in range(10)

        self.spectral_type = spectral_type
        self.number = number
        self.luminocity_class = luminocity_class

    def __str__(self):
        return f'{self.spectral_type}{self.number}{self.luminocity_class}'

    def __repr__(self):
        return f'Star.from_string(\'{str(self)}\')'

    @staticmethod
    def from_string(string):
        """Construct a new Star from a string describing the stellar type."""
        return Star(string[0], int(string[1]), string[2:])
```

(continues on next page)

(continued from previous page)

```
    def __serialize__(self, pack):
        return pack(str(self))

    @classmethod
    def __construct__(cls, data):
        return Star.from_string(data)
```

The class became quite a bit bigger. However, the `__str__`, `__repr__` and `from_string` methods are part of an interface you'd normally implement to make your class more useful.

[8]:
```
sun = Star('G', 2, 'V')
print("The Sun is a", sun, "type star.")
```

```
The Sun is a G2V type star.
```

[9]:
```
encoded_star = reg.to_json(sun, indent=2)
display_text(encoded_star, [4])
```

```
<IPython.core.display.HTML object>
```

The `__serialize__` method takes one argument (besides `self`). The argument `pack` is a function that creates the data record with all handles attached. The reason for this construct is that it takes keyword arguments for special cases.

```
def pack(data, ref=None, files=None):
    pass
```

- The `ref` argument, if given as `True`, will make sure that this object will not get reconstructed unnecessarily. One instance where this is incredibly useful, is if the object is a gigabytes large Numpy array.
- The `files` argument, when given, should be a list of filenames. This makes sure Noodles knows about the involvement of external files.

The data passed to `pack` maybe of any type, as long as the serialisation registry knows how to serialise it.

The `__construct__` method must be a *class method*. The `data` argument it is given can be expected to be identical to the data passed to the `pack` function at serialisation.

[10]:
```
decoded_star = reg.from_json(encoded_star)
display_text(repr(decoded_star))
```

```
<IPython.core.display.HTML object>
```

Writing a Serialiser class (example with large data)

Often, the class that needs serialising is not from your own package. In that case we need to write a specialised `Serialiser` class. For this purpose it may be nice to see how to serialise a Numpy array. This code is already in Noodles; we will look at a trimmed down version.

Given a NumPy array, we need to do two things:

- Generate a token by which to identify the array; we will use a SHA-256 hash to do this.
- Store the array effeciently; the HDF5 fileformat is perfectly suited.

SHA-256

We need to hash the combination of datatype, array shape and the binary data:

[11]:
```
import numpy
import hashlib
import base64

def array_sha256(a):
    """Create a SHA256 hash from a Numpy array."""
    dtype = str(a.dtype).encode()
    shape = numpy.array(a.shape)
    sha = hashlib.sha256()
    sha.update(dtype)
    sha.update(shape)
    sha.update(a.tobytes())
    return base64.urlsafe_b64encode(sha.digest()).decode()
```

Is this useable for large data? Let's see how this scales (code to generate this plot is below):

So on my laptop, hashing an array of ~1 GB takes a little over three seconds, and it scales almost perfectly linear. Next we define the storage routine (and a loading routine, but that's a oneliner).

[12]:
```
import h5py

def save_array_to_hdf5(filename, lock, array):
    """Save an array to a HDF5 file, using the SHA-256 of the array
    data as path within the HDF5. The `lock` is needed to prevent
    simultaneous access from multiple threads."""
    hdf5_path = array_sha256(array)
    with lock, h5py.File(filename) as hdf5_file:
        if not hdf5_path in hdf5_file:
            dataset = hdf5_file.create_dataset(
                hdf5_path, shape=array.shape, dtype=array.dtype)
            dataset[...] = array
            hdf5_file.close()

    return hdf5_path
```

And put it all together in a class derived from `SerArray`.

[13]:
```
import filelock
from noodles.serial import Serialiser, Registry


class SerArray(Serialiser):
    """Serialises Numpy array to HDF5 file."""
    def __init__(self, filename, lockfile):
        super().__init__(numpy.ndarray)
        self.filename = filename
        self.lock = filelock.FileLock(lockfile)

    def encode(self, obj, pack):
        key = save_array_to_hdf5(self.filename, self.lock, obj)
        return pack({
            "filename": self.filename,
```

(continues on next page)

(continued from previous page)

```
            "hdf5_path": key,
        }, files=[self.filename], ref=True)

    def decode(self, cls, data):
        with self.lock, h5py.File(self.filename) as hdf5_file:
            return hdf5_file[data["hdf5_path"]].value
```

We have to insert the serialiser into a new registry.

[14]:
```
!rm -f tutorial.h5  # remove from previous run
```

[15]:
```
import noodles
from noodles.tutorial import display_text

def registry():
    return Registry(
        parent=noodles.serial.base(),
        types={
            numpy.ndarray: SerArray('tutorial.h5', 'tutorial.lock')
        })

reg = registry()
```

Now we can serialise our first Numpy array!

[16]:
```
encoded_array = reg.to_json(numpy.arange(10), host='localhost', indent=2)
display_text(encoded_array, [6])
```

```
<IPython.core.display.HTML object>
```

Now, we should be able to read back the data directly from the HDF5.

[17]:
```
with h5py.File('tutorial.h5') as f:
    result = f['4Z8kdMg-CbjgTKKYlz6b-_-Tsda5VAJL44OheRB10mU='].value
print(result)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

We have set the `ref` property to `True`, we can now read back the serialised object without dereferencing. This will result in a placeholder object containing only the encoded data:

[18]:
```
ref = reg.from_json(encoded_array)
display_text(ref)
display_text(vars(ref), max_width=60)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

If we want to retrieve the data we should run `from_json` with `deref=True`:

[19]:
```
display_text(reg.from_json(encoded_array, deref=True))
```

```
<IPython.core.display.HTML object>
```

Appendix A: better parsing

If you're interested in doing a bit better in parsing generic expressions into objects, take a look at `pyparsing`.

[20]:
```
!pip install pyparsing
```

```
Requirement already satisfied: pyparsing in /home/johannes/.local/share/workon/
↪noodles/lib/python3.6/site-packages
```

The following code will parse the stellar types we used before:

[21]:
```
from pyparsing import Literal, replaceWith, OneOrMore, Word, nums, oneOf

def roman_numeral_literal(string, value):
    return Literal(string).setParseAction(replaceWith(value))

one = roman_numeral_literal("I", 1)
four = roman_numeral_literal("IV", 4)
five = roman_numeral_literal("V", 5)

roman_numeral = OneOrMore(
    (five | four | one).leaveWhitespace()) \
    .setName("roman") \
    .setParseAction(lambda s, l, t: sum(t))

integer = Word(nums) \
    .setName("integer") \
    .setParseAction(lambda t:int(t[0]))

mkStar = oneOf(list("OBAFGKM")) + integer + roman_numeral
```

[22]:
```
list(mkStar.parseString('B2IV'))
```

[22]:
```
['B', 2, 4]
```

[23]:
```
roman_class = {
    'I': 'supergiant',
    'II': 'bright giant',
    'III': 'regular giant',
    'IV': 'sub-giants',
    'V': 'main-sequence',
    'VI': 'sub-dwarfs',
    'VII': 'white dwarfs'
}
```

Appendix B: measuring SHA-256 performance

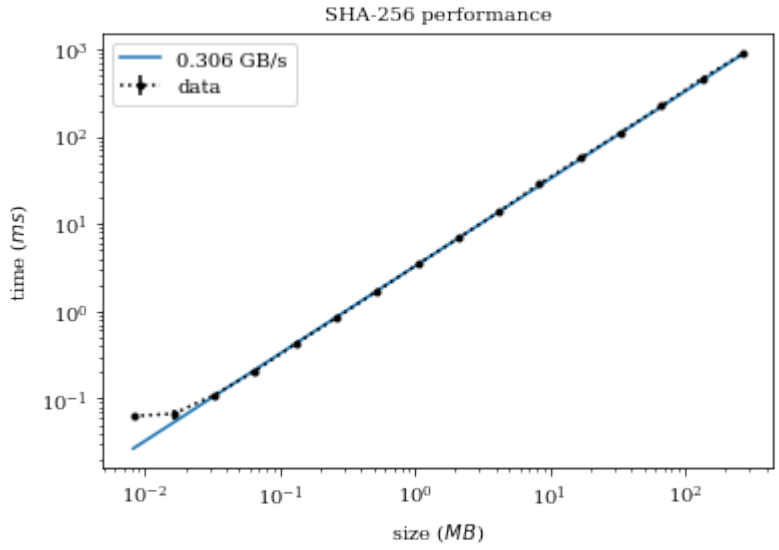
[24]:
```
import timeit
import matplotlib.pyplot as plt
plt.rcParams['font.family'] = "serif"
from scipy import stats


def benchmark(size, number=10):
    """Measure performance of SHA-256 hashing large arrays."""
    data = numpy.random.uniform(size=size)
    return timeit.timeit(
        stmt=lambda: array_sha256(data),
        number=number) / number
```

(continues on next page)

(continued from previous page)

```
sizes = numpy.logspace(10, 25, 16, base=2, dtype=int)
timings = numpy.array([[benchmark(size, 1) for size in sizes]
                       for i in range(10)])

sizes_MB = sizes * 8 / 1e6
timings_ms = timings.mean(axis=0) * 1000
timings_err = timings.std(axis=0) * 1000

slope, intercept, _, _, _ = stats.linregress(
    numpy.log(sizes_MB[5:]),
    numpy.log(timings_ms[5:]))

print("scaling:", slope, "(should be ~1)")
print("speed:", numpy.exp(-intercept), "GB/s")

ax = plt.subplot(111)
ax.set_xscale('log', nonposx='clip')
ax.set_yscale('log', nonposy='clip')
ax.plot(sizes_MB, numpy.exp(intercept) * sizes_MB,
        label='{:.03} GB/s'.format(numpy.exp(-intercept)))
ax.errorbar(sizes_MB, timings_ms, yerr=timings_err,
            marker='.', ls=':', c='k', label='data')
ax.set_xlabel('size ($MB$)')
ax.set_ylabel('time ($ms$)')
ax.set_title('SHA-256 performance', fontsize=10)
ax.legend()
plt.savefig('sha256-performance.svg')
plt.show()
```

```
scaling: 1.0064819484 (should be ~1)
speed: 0.305958896153 GB/s
```



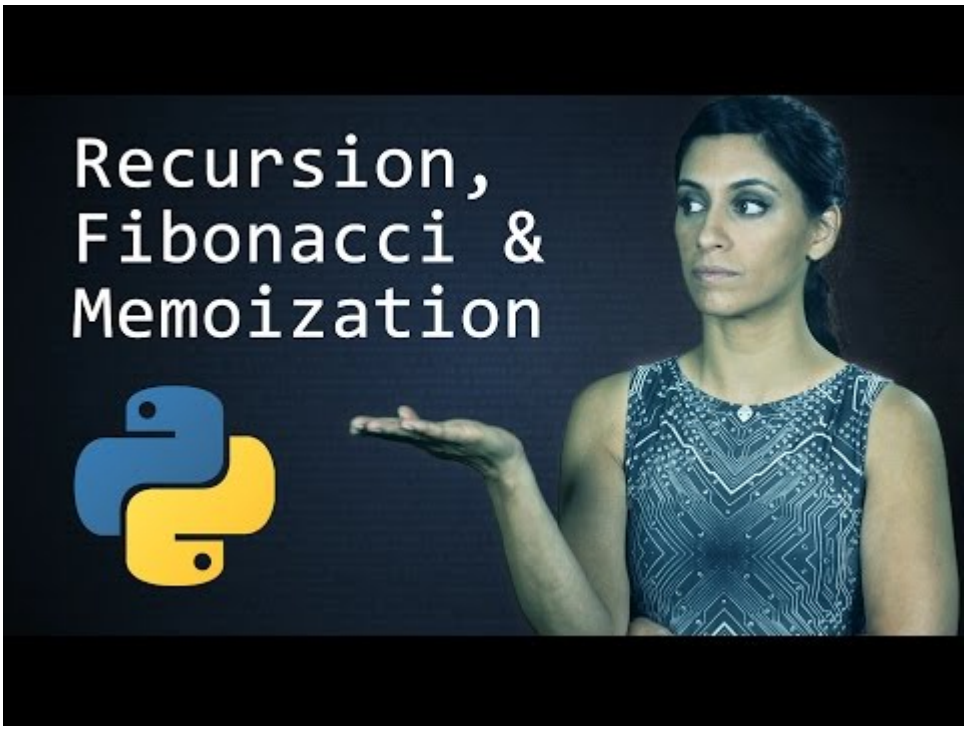
Implementation

A `Registry` object roughly consists of three parts. It works like a dictionary searching for `Serialisers` based on the class or baseclass of an object. If an object cannot be identified through its class or baseclasses the `Registry` has a function hook that may use any test to determine the proper `Serialiser`. When neither the hook nor the dictionary give a result, there is a default fall-back option.

4.3.7 Advanced: Control your flow

Here we dive a bit deeper in advanced flow control in Noodles. Starting with a recap into for-loops, moving on to conditional evaluation of workflows and standard algorithms. This chapter will also go a bit deeper into the teritory of functional programming. Specifically, we will see how to program sequential loops using only functions and recursion.

If you are new to the concepts of *recursion*, here is some nice material to start with:



Recap: for loops

In the *Translating Poetry* tutorial we saw how we could create parallel `for` loops in Noodles. To recap, let's reverse the words in a sentence. Assume you have the following for-loop in Python:

[1]:
```
sentence = 'the quick brown fox jumps over the lazy dog'
reverse = []

def reverse_word(word):
    return word[::-1]

for word in sentence.split():
    reverse.append(reverse_word(word))
```

(continues on next page)

(continued from previous page)

```
result = ' '.join(reverse)
print(result)
```

```
eht kciuq nworb xof spmuj revo eht yzal god
```

There is a pattern to this code that is better written as:

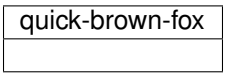
[2]:
```
reverse = [reverse_word(word) for word in sentence.split()]
result = ' '.join(reverse)
print(result)
```

```
eht kciuq nworb xof spmuj revo eht yzal god
```

This last version can be translated to Noodles. Assume for some reason we want to schedule the `reverse_word` function (it takes forever to run on a single core!). Because `reverse_words` becomes a *promise*, the line with `' '.join(reverse)` also has to be captured in a scheduled function.

[3]:
```
import noodles

@noodles.schedule
def reverse_word(word):
    return word[::-1]

@noodles.schedule
def make_sentence(words):
    return ' '.join(words)

reverse_words = noodles.gather_all(
    reverse_word(word) for word in sentence.split())
workflow = make_sentence(reverse_words)
```

[4]:
```
from noodles.tutorial import display_workflows
noodles.tutorial.display_workflows(prefix='control', quick_brown_fox=workflow)
```



This example shows how we can do loops in parallel. There are cases where we will need to do loops in a serialised manner. For example, if we are handling a very large data set and all of the computation does not fit in memory when done in parallel.

There are hybrid *divide and conquer* approaches that can be implemented in Noodles. We then chunk all the work in blocks that can be executed in parallel, and stop when the first chunk gives us reason to. Divide-and-conquer can be implemented using a combination of the two looping strategies (parallel and sequential).

Sequential loops are made using recursion techniques.

Recursion

Sequential loops can be made in Noodles using recursion. Comes the obligatory factorial function example:

[16]:
```
from noodles.tutorial import display_text

def factorial(x):
    if x == 0:
        return 1
```

(continues on next page)

(continued from previous page)

```
    else:
        return factorial(x - 1) * x

display_text('100! = {}'.format(factorial(100)))
```

```
<IPython.core.display.HTML object>
```

There is a problem with such a recursive algorithm when numbers get too high.

[18]:
```
try:
    display_text('10000! =', factorial(10000))
except RecursionError as e:
    display_text(e)
```

```
<IPython.core.display.HTML object>
```

Yikes! Let's head on. And translate the program to Noodles. Suppose we make `factorial` a scheduled function, we cannot multiply a *promise* with a *number* just like that (at least not in the current version of Noodles). We change the function slightly with a second argument that keeps count. This also makes the `factorial` function *tail-recursive*.

[19]:
```
@noodles.schedule
def factorial(x, acc=1):
    if x == 0:
        return acc
    else:
        return factorial(x - 1, acc * x)

result = noodles.run_single(factorial(10000))

display_text('10000! = {}'.format(result))
```

```
<IPython.core.display.HTML object>
```

Yeah! Noodles runs the tail-recursive function iteratively! This is actually **very important**. We'll do a little experiment. Start your system monitor (plotting a graph of your memory usage) and run the following snippets. We let every function call to factorial gobble up some memory and to be able to measure the effect of that we insert a small sleep. Fair warning: With the current setting of `gobble_size` and running 50 loops, the first version will take about **4GB** of memory. Just change the size so that a measurable fraction of your RAM is taken up by the process and you can see the result.

[37]:
```
import numpy
import time

gobble_size = 10000000
```

[89]:
```
@noodles.schedule(call_by_ref=['gobble'])
def mul(x, y, gobble):
    return x*y

@noodles.schedule(call_by_ref=['gobble'])
def factorial(x, gobble):
    time.sleep(0.1)
    if x == 0:
        return 1
    else:
        return mul(factorial(x - 1, copy(gobble)), x, gobble)
```

(continues on next page)

(continued from previous page)

```
gobble = numpy.zeros(gobble_size)
result = noodles.run_single(factorial(50, gobble))
```

We passed the `gobble` argument by reference. This prevents Noodles from copying the array when creating the workflow. If you have functions that take large arrays as input *and you don't change the value of the array in between calls* this is a sensible thing to do. On my machine, running only 10 loops, this gives the following result:

Try to understand why this happens. We have reserved a NumPy array with `gobble_size` (10^7) floating points of 8 bytes each. The total size in bytes of this array is $8 \times 10^7\ MB$. In each recursive call to `factorial` the array is copied, so in total this will use $10 \cdot 8 \times 10^7\ MB = 800\ MB$ of memory!

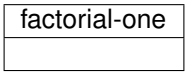
The next version is **tail-recursive**. This should barely make a dent in your memory usage!

[98]:
```
@noodles.schedule(call_by_ref=['gobble'])
def factorial_tr(x, acc=1, gobble=None):
    time.sleep(0.1)
    if x == 0:
        return acc
    else:
        return factorial_tr(x - 1, mul(acc, x, gobble), copy(gobble))

gobble = np.zeros(gobble_size)
result = noodles.run_single(factorial_tr(50, gobble=gobble))
```

Now, the `factorial` function is still recursive. However, since returning a call to the `factorial` function is last thing we do, the intermediate results can be safely thrown away. We'll have in memory the original reference to `gobble` and one version in the Noodles run-time for the last time `factorial` returned a workflow where `gobble.copy()` was one of the arguments. In total this gives a memory consumption of 160 MB (plus a little extra for the Python run-time itself). We see peeks that reach over 250 MB in the graph: this is where `gobble` is being copied, after which the garbage collector deletes the old array.

Try to understand why this happens. In the first case the function returns a new workflow to be evaluated. This workflow has two nodes:

[99]:
```
display_workflows(
    prefix='control',
    factorial_one=noodles.unwrap(factorial)(10, '<memory gobble>'))
```



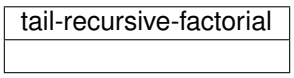
To evaluate this workflow, Noodles first runs the top node `factorial(9, '<memory gobble>')`. When the answer for this function is obtained it is inserted into the slot for `mul(-, 10)`. Until the entire workflow is evaluated, the `<memory gobble>` remains in memory. Before this happens the factorial function is called which copies the

gobble and creates a new workflow! We can write this out by expanding our algorithm symbolically $f(x) = x \cdot f(x-1)$:

$$
\begin{aligned}
f(10) &= 10 \cdot f(9) && (4.1)\\
&= 10 \cdot (9 \cdot f(8)) && (4.2)\\
&= 10 \cdot (9 \cdot (8 \cdot f(7))) && (4.3)\\
& && (4.4)\\
&= 10 \cdot (9 \cdot (8 \cdot (7 \cdot (6 \cdot (5 \cdot (4 \cdot (3 \cdot (2 \cdot 1))))))) && (4.5)\\
&= 10 \cdot (9 \cdot (8 \cdot (7 \cdot (6 \cdot (5 \cdot (4 \cdot (3 \cdot 2)))))) && (4.6)\\
&= 10 \cdot (9 \cdot (8 \cdot (7 \cdot (6 \cdot (5 \cdot (4 \cdot 6)))))) && (4.7)\\
& && (4.8)
\end{aligned}
$$

Now for the tail-recursive version, the workflow looks a bit different:

[100]:
```
display_workflows(
    prefix='control',
    tail_recursive_factorial=noodles.unwrap(factorial_tr)(10, gobble='<memory gobble>
↪'))
```



First the `mul(1, 10, '<memory gobble>')` is evaluated. Its result is inserted into the empty slot in the call to `factorial_tr`. This call returns a new workflow with a new copy of `<memory gobble>`. This time however, the old workflow can be safely deleted. Again, it helps to look at the algorithm symbolically, given $f(x, a) = f(x-1, x \cdot a)$:

$$
\begin{aligned}
f(10, 1) &= f(9, (10 \cdot 1)) && (4.9)\\
&= f(8, (9 \cdot 10)) && (4.10)\\
&= f(8, (9 \cdot 10)) && (4.11)\\
&= f(7, (8 \cdot 90)) && (4.12)\\
& && (4.13)\\
& && (4.14)
\end{aligned}
$$

Conditional evaluation

But Python has more statements for flow control! The conditional execution of code is regulated through the `if` statement. You may want to make the exection of parts of your workflow conditional based on intermediate results. One such instance may look like this:

[5]:
```
@noodles.schedule
def method_one(x):
    pass

@noodles.schedule
def method_two(x):
    pass

@noodles.schedule
def what_to_do(x):
    if condition(x):
        return method_one(x)
    else:
        return method_two(x)
```

We've put the `if`-statement inside the scheduled function `what_to_do`. This returns a new workflow depending on the value of `x`. We can no longer get a nice single graph picture of the workflow, because the workflow doesn't exist! (there is no spoon ...) We can work through a small example from the Python tutorial: computing prime numbers.

[6]:
```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

The core computation in this example is the `n % x == 0` bit. So we start by creating a scheduled function that does that.

[7]:
```
@noodles.schedule
def divides(n, x):
    return n % x == 0
```

Noodles can parallelize the inner loop, but this gives a problem: how do we know when to stop? There is no way to get it both ways.

First, we'll see how to do the parallel solution. We'll compute the `divides(n, x)` function for the values of `n` and `x` and then filter out those where `divides` gave `False`. This last step is done using the `compress` function.

[8]:
```
@noodles.schedule
def compress(lst):
    """Takes a list of pairs, returns a list of
    first elements of those pairs for which the
    second element is thruthy."""
    return [a for a, b in lst if b]
```

Using the `compress` function we can write the Noodlified parallel version of the `filter` function. We'll call it `p_filter` for *parallel filter*.

[9]:
```
?filter
```

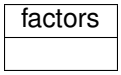
```
[0;31mInit signature:[0m [0mfilter[0m[0;34m([0m[0mself[0m[0;34m,[0m [0;34m/[0m[0;34m,
↪[0m [0;34m*[0m[0margs[0m[0;34m,[0m [0;34m**[0m[0mkwargs[0m[0;34m)[0m[0;34m[0m
[0;31mDocstring:[0m
filter(function or None, iterable) --> filter object

Return an iterator yielding those items of iterable for which function(item)
is true. If function is None, return the items that are true.
[0;31mType:[0m           type
```

Using the generic `p_filter` function we then write the function `find_factors` that finds all integer factors of a number in parallel. Both `p_filter` and `find_factors` won't be scheduled functions. Rather, together they build

the workflow that solves our problem.

[10]:
```
def p_filter(f, lst):
    return compress(noodles.gather_all(
        noodles.gather(x, f(x)) for x in lst))

def find_factors(n):
    return p_filter(lambda x: divides(n, x), range(2, n))
```

[11]:
```
display_workflows(prefix='control', factors=find_factors(5))
```



No we can run this workflow for all the numbers we like.

[12]:
```
result = noodles.run_parallel(
    noodles.gather_all(noodles.gather(n, find_factors(n))
                       for n in range(2, 10)),
    n_threads=4)

for n, factors in result:
    if factors:
        print(n, 'equals', ', '.join(
            '{}*{}'.format(x, n//x) for x in factors))
    else:
        print(n, 'is prime')
```

```
2 is prime
3 is prime
4 equals 2*2
5 is prime
6 equals 2*3, 3*2
7 is prime
8 equals 2*4, 4*2
9 equals 3*3
```

Few! We managed, but if all we wanted to do is find primes, we did way too much work; we also found all factors of the numbers. We had to write some boiler plate code. Argh, this tutorial was supposed to be on flow control! We move on to the sequential version. Wait, I hear you think, we were using Noodles to do things in parallel!?? Why make an effort to do sequential work? Well, we'll need it to implement the divide-and-conquer strategy, among other things. Noodles is not only a framework for parallel programming, but it also works concurrent. In the context of a larger workflow we may still want to make decision steps on a sequential basis, while another component of the workflow is happily churning out numbers.

Find-first

Previously we saw the definition of a Noodlified `filter` function. How can we write a `find_first` that stops after finding a first match? If we look at the workflow that `p_filter` produces, we see that all predicates are already present in the workflow and will be computed concurrently. We now write a sequential version. We may achieve sequential looping through recursion like this:

[19]:
```
def find_first(f, lst):
    if not lst:
        return None
    elif f(lst[0]):
```

(continues on next page)

(continued from previous page)

```
        return lst[0]
    else:
        return find_first(f, lst[1:])
```

However, if `f` is a scheduled function `f(lst[0])` will give a promise, and this routine will fail.

[20]:
```
@noodles.schedule
def find_first_helper(f, lst, first):
    if first:
        return lst[0]
    elif len(lst) == 1:
        return None
    else:
        return find_first_helper(f, lst[1:], f(lst[1]))

def find_first(f, lst):
    return find_first_helper(f, lst, f(lst[0]))
```

[21]:
```
noodles.run_single(find_first(lambda x: divides(77, x), range(2, 63)))
```

[21]:
```
7
```

That works. Now suppose the input list is somewhat harder to compute; every element is the result of a workflow.

Appendix: creating memory profile plots

[83]:
```
%%writefile test-tail-recursion.py
import numpy
import noodles
import time
from copy import copy

@noodles.schedule(call_by_ref=['gobble'])
def factorial_tr(x, acc=1, gobble=None):
    time.sleep(0.1)
    if x == 0:
        return acc
    else:
        return factorial_tr(x - 1, acc * x, copy(gobble))

gobble_size = 10000000
gobble = numpy.zeros(gobble_size)
result = noodles.run_single(factorial_tr(10, gobble=gobble))
```

```
Overwriting test-tail-recursion.py
```

[90]:
```
%%writefile test-recursion.py
import numpy
import noodles
import time
from copy import copy

@noodles.schedule(call_by_ref=['gobble'])
def mul(x, y, gobble):
    return x*y
```

(continues on next page)

(continued from previous page)

```
@noodles.schedule(call_by_ref=['gobble'])
def factorial(x, gobble):
    time.sleep(0.1)
    if numpy.all(x == 0):
        return numpy.ones_like(x)
    else:
        return mul(factorial(x - 1, copy(gobble)), x, gobble)

gobble_size = 10000000
gobble = numpy.zeros(gobble_size)
result = noodles.run_single(factorial(10, gobble))
```

```
Overwriting test-recursion.py
```

[91]:
```
!pip install memory_profiler
```

```
Requirement already satisfied: memory_profiler in /home/johannes/.local/share/workon/
↪noodles/lib/python3.6/site-packages
Requirement already satisfied: psutil in /home/johannes/.local/share/workon/noodles/
↪lib/python3.6/site-packages (from memory_profiler)
```

[92]:
```
%%bash
rm mprofile_*.dat
mprof run -T 0.001 python ./test-tail-recursion.py
mprof run -T 0.001 python ./test-recursion.py
```

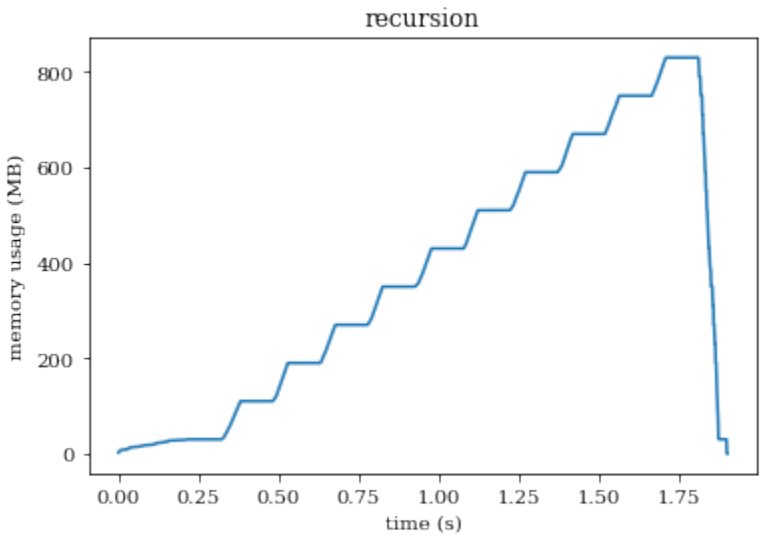
```
mprof: Sampling memory every 0.001s
mprof: Sampling memory every 0.001s
```

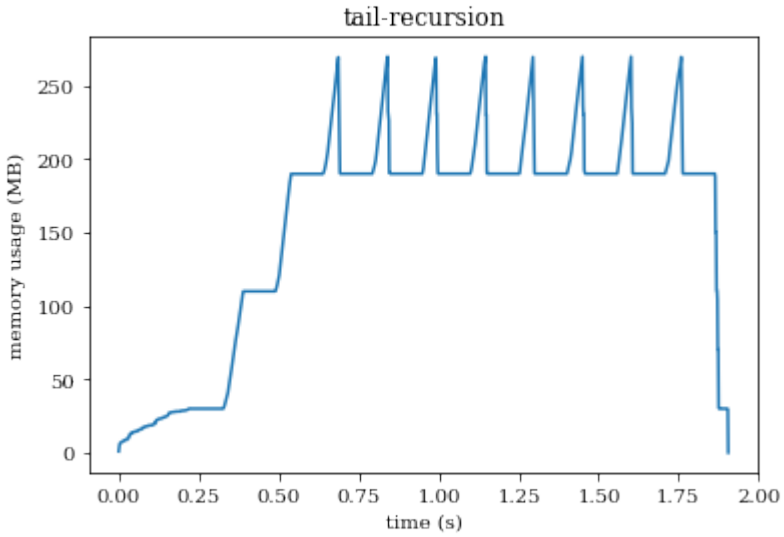
[93]:
```
from pathlib import Path
from matplotlib import pyplot as plt

plt.rcParams['font.family'] = 'serif'

def read_mprof(filename):
    lines = list(open(filename, 'r'))
    cmd = filter(lambda l: l[:3] == 'CMD', lines)
    mem = filter(lambda l: l[:3] == 'MEM', lines)
    data = np.array([list(map(float, l.split()[1:])) for l in mem])
    data[:,1] -= data[0,1]
    data[:,0] *= 1024**2
    return cmd, data

def plot_mprof(filename):
    cmd, data = read_mprof(filename)
    if 'tail' in next(cmd):
        figname = 'tail-recursion'
    else:
        figname = 'recursion'

    plt.plot(data[:,1], data[:,0] / 1e6)
    plt.xlabel('time (s)')
    plt.ylabel('memory usage (MB)')
    plt.title(figname)
    plt.savefig('control-' + figname + '-raw.svg', bbox_inches='tight')
    plt.show()
```

(continues on next page)

(continued from previous page)

```
files = list(Path('.').glob('mprofile_*.dat'))
for f in files:
    plot_mprof(f)
    plt.close()
```





[ ]:

4.4 Implementation

4.4.1 Development documentation

(stub)

Noodles

noodles.**schedule**(*f*, ***hints*)
Decorator; schedule calls to function `f` into a workflow, in stead of running them at once. The decorated function returns a `PromisedObject`.

noodles.**schedule_hint**(***hints*)
Decorator; same as *schedule()*, with added hints. These hints can be anything.

noodles.**run_single**(*workflow*)
"Run workflow in a single thread (same as the scheduler).

Parameters **workflow** – Workflow or PromisedObject to be evaluated.

Returns Evaluated result.

noodles.**run_process**(*workflow*, *, *n_processes*, *registry*, *verbose=False*, *jobdirs=False*, *init=None*, *finish=None*, *deref=False*)
Run the workflow using a number of new python processes. Use this runner to test the workflow in a situation where data serial is needed.

Parameters

- **workflow** (`Workflow` or `PromisedObject`) – The workflow.
- **n_processes** – Number of processes to start.
- **registry** – The serial registry.
- **verbose** – Request verbose output on worker side
- **jobdirs** – Create a new directory for each job to prevent filename collision.(NYI)
- **init** – An init function that needs to be run in each process before other jobs can be run. This should be a scheduled function returning True on success.
- **finish** – A function that wraps up when the worker closes down.
- **deref** (*bool*) – Set this to True to pass the result through one more encoding and decoding step with object derefencing turned on.

Returns the result of evaluating the workflow

Return type any

class noodles.**Scheduler**(*verbose=False*, *error_handler=None*, *job_keeper=None*)
Schedules jobs, recieves results, then schedules more jobs as they become ready to compute. This class communicates with a pool of workers by means of coroutines.

run(*connection: noodles.lib.connection.Connection*, *master: noodles.workflow.model.Workflow*)
Run a workflow.

Parameters

- **connection** (`Connection`) – A connection giving a sink to the job-queue and a source yielding results.
- **master** (`Workflow`) – The workflow.

noodles.**has_scheduled_methods**(*cls*)
Decorator; use this on a class for which some methods have been decorated with *schedule()* or *schedule_hint()*. Those methods are then tagged with the attribute `__member_of__`, so that we may serialise and retrieve the correct method. This should be considered a patch to a flaw in the Python object model.

class noodles.**Fail**(*func*, *fails=None*, *exception=None*)
Signifies a failure in a computation that was wrapped by a `@maybe` decorator. Because Noodles runs all functions from the same context, it is not possible to use Python stack traces to find out where an error happened. In stead we use a `Fail` object to store information about exceptions and the subsequent continuation of the failure.

add_call(*func*)
Add a call to the trace.

is_root_cause
If the field `exception` is set in this object, it means that we are looking at the root cause of the failure.

noodles.**failed**(*obj*)
Returns True if `obj` is an instance of `Fail`.

noodles.**run_logging**(*wf*, *n_threads*, *display*)
Adds a display to the parallel runner. Because messages come in asynchronously now, we start an extra thread just for the display routine.

noodles.**run_parallel**(*workflow*, *n_threads*)
Run a workflow in parallel threads.

Parameters

- **workflow** – Workflow or PromisedObject to evaluate.
- **n_threads** – number of threads to use (in addition to the scheduler).

Returns evaluated workflow.

noodles.**unwrap**(*f*)
Safely obtain the inner function of a previously wrapped (or decorated) function. This either returns `f.__wrapped__` or just `f` if the latter fails.

noodles.**gather**(**a*)
(scheduled) Converts a list of promises (i.e. `PromisedObject`) to a promised list of values.

noodles.**gather_all**(*a*)
Converts an iterator of promises into a promise of a list.

noodles.**gather_dict**(***kwargs*)
(scheduled) Creates a promise of a dictionary.

noodles.**lift**(*obj*, *memo=None*)
Make a promise out of object `obj`, where `obj` may contain promises internally.

Parameters

- **obj** – Any object.
- **memo** – used for internal caching (similar to `deepcopy()`).

If the object is a `PromisedObject`, or *pass-by-value* (`str`, `int`, `float`, `complex`) it is returned as is.

If the object's `id` has an entry in `memo`, the value from `memo` is returned.

If the object has a method `__lift__`, it is used to get the promise. `__lift__` should take one additional argument for the `memo` dictionary, entirely analogous to `deepcopy()`.

If the object is an instance of one of the basic container types (list, dictionary, tuple and set), we use the analogous function (`make_list()`, `make_dict()`, `make_tuple()`, and `make_set()`) to promise their counterparts should these objects contain any promises. First, we map all items in the container through *`lift()`*, then check the result for any promises. Note that in the case of dictionaries, we lift all the items (i.e. the list of key/value tuples) and then construct a new dictionary.

If the object is an instance of a subclass of any of the basic container types, the `__dict__` of the object is lifted as well as the object cast to its base type. We then use `set_dict()` to set the `__dict__` of the new promise. Again, if the object did not contain any promises, we return it without change.

Otherwise, we lift the `__dict__` and create a promise of a new object of the same class as the input, using `create_object()`. This works fine for what we call *reasonable* objects. Since calling *lift()* is an explicit action, we do not require reasonable objects to be derived from `Reasonable` as we do with serialisation, where such a default behaviour could lead to unexplicable bugs.

`noodles.`**`unpack`**(*t*, *n*)
Iterates over a promised sequence, the sequence should support random access by `object.__getitem__()`. Also the length of the sequence should be known beforehand.

Parameters

- **`t`** – a sequence.
- **`n`** – the length of the sequence.

Returns an unpackable generator for the elements in the sequence.

`noodles.`**`maybe`**(*func*)
Calls `f` in a try/except block, returning a *Fail* object if the call fails in any way. If any of the arguments to the call are Fail objects, the call is not attempted.

`noodles.`**`delay`**(*value*)
(scheduled) Creates a promise of a given value. TODO: this function should have a different name.

`noodles.`**`update_hints`**(*obj*, *data*)
Update the hints on the root-node of a workflow. Usually, schedule hints are fixed per function. Sometimes a user may want to set hints manually on a specific promised object. *update_hints()* uses the `update` method on the hints dictionary with `data` as its argument.

Parameters

- **`obj`** – a `PromisedObject`.
- **`data`** – a `dict` containing additional hints.

The hints are modified, in place, on the node. All workflows that contain the node are affected.

`noodles.`**`quote`**(*promise*)
Quote a promise.

`noodles.`**`unquote`**(*quoted*)
Unquote a quoted promise.

`noodles.`**`result`**(*obj*)
Results are stored on the nodes in the workflow at run time. This function can be used to get at a result of a node in a workflow after run time. This is not a recommended way of getting at results, but can help with debugging.

`noodles.`**`fold`**(*fun: Callable*, *state: Any*, *xs: Iterable*)
(scheduled) Traverse an iterable object while performing stateful computations with the elements. It returns a `PromisedObject` containing the result of the stateful computations.

For a general definition of folding see: https://en.wikipedia.org/wiki/Fold_(higher-order_function)

Parameters

- **`fun`** – stateful function.
- **`state`** – initial state.
- **`xs`** – iterable object.

Returns PromisedObject

noodles.**find_first**(*pred*, *lst*)
Find the first result of a list of promises `lst` that satisfies a predicate `pred`.

Parameters

- **pred** – a function of one argument returning `True` or `False`.
- **lst** – a list of promises or values.

Returns a promise of a value or `None`.

This is a wrapper around `s_find_first()`. The first item on the list is passed *as is*, forcing evalutation. The tail of the list is quoted, and only unquoted if the predicate fails on the result of the first promise.

If the input list is empty, `None` is returned.

noodles.**conditional**(*b: bool*, *branch_true: Any*, *branch_false: Any = None*) → Any
Control statement to follow a branch in workflow. Equivalent to the `if` statement in standard Python.

The quote function delay the evaluation of the branches until the boolean is evaluated.

Parameters

- **b** – promised boolean value.
- **branch_true** – statement to execute in case of a true predicate.
- **branch_false** – default operation to execute in case of a false predicate.

Returns PromisedObject

noodles.**simple_lift**(*obj*)
(scheduled) Create a promise from a plain object.

Internal Specs

noodles.workflow.**invert_links**(*links*)
Inverts the call-graph to get a dependency graph. Possibly slow, short version.

Parameters **links** (*Mapping[NodeId, Set[(NodeId, ArgumentType, [int|str])]]*) – forward links of a call-graph.

Returns inverted graph, giving dependency of jobs.

Return type Mapping[NodeId, Mapping[(ArgumentType, [int|str]), NodeId]]

noodles.workflow.**from_call**(*foo*, *args*, *kwargs*, *hints*, *call_by_value=True*)
Takes a function and a set of arguments it needs to run on. Returns a newly constructed workflow representing the promised value from the evaluation of the function with said arguments.

These arguments are stored in a BoundArguments object matching to the signature of the given function `foo`. That is, bound_args was constructed by doing:

```
inspect.signature(foo).bind(*args, **kwargs)
```

The arguments stored in the `bound_args` object are filtered on being either plain, or promised. If an argument is promised, the value it represents is not actually available and needs to be computed by evaluating a workflow.

If an argument is a promised value, the workflow representing the value is added to the new workflow. First all the nodes in the original workflow, if not already present in the new workflow from an earlier argument, are copied to the new workflow, and a new entry is made into the link dictionary. Then the links in the

old workflow are also added to the link dictionary. Since the link dictionary points from nodes to a `set` of *`ArgumentAddress`* es, no links are duplicated.

In the `bound_args` object the promised value is replaced by the `Empty` object, so that we can see which arguments still have to be evaluated.

Doing this for all promised value arguments in the bound_args object, results in a new workflow with all the correct dependencies represented as links in the graph.

Parameters

- **foo** (*`Callable`*) – Function (or object) being called.
- **args** – Normal arguments to call
- **kwargs** – Keyword arguments to call
- **hints** – Hints that can be passed to the scheduler on where or how to schedule this job.

Returns New workflow.

Return type *Workflow*

class `noodles.workflow.`**Workflow**(*root*, *nodes*, *links*)

The workflow data container.

root

A reference to the root node in the graph.

nodes

A `dict` listing the nodes in the graph. We use a `dict` only to have a persistent object reference.

links

A `dict` giving a `set` of links from each node.

class `noodles.workflow.`**FunctionNode**(*foo*, *bound_args*, *hints*, *result*)

Captures a function call as a combination of function and arguments. Some of these arguments may be set to *`Empty`*, these need to be filled in by the workflow before the function can be applied.

foo

The function (or object) that is being called.

bound_args

A `BoundArguments` object storing the arguments to the function.

data

Convert to a *`NodeData`* for subsequent serial.

class `noodles.workflow.`**NodeData**(*function*, *arguments*, *hints*)

arguments

Alias for field number 1

function

Alias for field number 0

hints

Alias for field number 2

`noodles.workflow.`**insert_result**(*node*, *address*, *value*)

Runs `set_argument`, but checks first wether the data location is not already filled with some data. In any normal circumstance this checking is redundant, but if we don't give an error here the program would continue with unexpected results.

noodles.workflow.**Empty**
alias of inspect._empty

noodles.workflow.**is_node_ready**(*node*)
Returns True if none of the argument holders contain any *Empty* object.

class noodles.workflow.**Argument**(*address*, *value*)

address
Alias for field number 0

value
Alias for field number 1

class noodles.workflow.**ArgumentAddress**(*kind*, *name*, *key*)
Codifies a value given for some argument.

key
Alias for field number 2

kind
Alias for field number 0

name
Alias for field number 1

class noodles.workflow.**ArgumentKind**
Codifies the location to a unique argument of a function.

Promised object

noodles.interface.**delay**(*value*)
(scheduled) Creates a promise of a given value. TODO: this function should have a different name.

noodles.interface.**gather**(**a*)
(scheduled) Converts a list of promises (i.e. *PromisedObject*) to a promised list of values.

noodles.interface.**gather_all**(*a*)
Converts an iterator of promises into a promise of a list.

noodles.interface.**gather_dict**(***kwargs*)
(scheduled) Creates a promise of a dictionary.

noodles.interface.**schedule_hint**(***hints*)
Decorator; same as *schedule()*, with added hints. These hints can be anything.

noodles.interface.**schedule**(*f*, ***hints*)
Decorator; schedule calls to function `f` into a workflow, in stead of running them at once. The decorated function returns a *PromisedObject*.

noodles.interface.**unpack**(*t*, *n*)
Iterates over a promised sequence, the sequence should support random access by `object.__getitem__()`. Also the length of the sequence should be known beforehand.

Parameters

- **t** – a sequence.
- **n** – the length of the sequence.

Returns an unpackable generator for the elements in the sequence.

noodles.interface.**has_scheduled_methods**(*cls*)
Decorator; use this on a class for which some methods have been decorated with *schedule()* or *schedule_hint()*. Those methods are then tagged with the attribute __member_of__, so that we may serialise and retrieve the correct method. This should be considered a patch to a flaw in the Python object model.

noodles.interface.**unwrap**(*f*)
Safely obtain the inner function of a previously wrapped (or decorated) function. This either returns `f.__wrapped__` or just `f` if the latter fails.

noodles.interface.**update_hints**(*obj*, *data*)
Update the hints on the root-node of a workflow. Usually, schedule hints are fixed per function. Sometimes a user may want to set hints manually on a specific promised object. *update_hints()* uses the `update` method on the hints dictionary with `data` as its argument.

Parameters

- **obj** – a *PromisedObject*.
- **data** – a `dict` containing additional hints.

The hints are modified, in place, on the node. All workflows that contain the node are affected.

noodles.interface.**lift**(*obj*, *memo=None*)
Make a promise out of object `obj`, where `obj` may contain promises internally.

Parameters

- **obj** – Any object.
- **memo** – used for internal caching (similar to `deepcopy()`).

If the object is a *PromisedObject*, or *pass-by-value* (`str`, `int`, `float`, `complex`) it is returned as is.

If the object's `id` has an entry in `memo`, the value from `memo` is returned.

If the object has a method `__lift__`, it is used to get the promise. `__lift__` should take one additional argument for the `memo` dictionary, entirely analogous to `deepcopy()`.

If the object is an instance of one of the basic container types (list, dictionary, tuple and set), we use the analogous function (`make_list()`, `make_dict()`, `make_tuple()`, and `make_set()`) to promise their counterparts should these objects contain any promises. First, we map all items in the container through *lift()*, then check the result for any promises. Note that in the case of dictionaries, we lift all the items (i.e. the list of key/value tuples) and then construct a new dictionary.

If the object is an instance of a subclass of any of the basic container types, the `__dict__` of the object is lifted as well as the object cast to its base type. We then use `set_dict()` to set the `__dict__` of the new promise. Again, if the object did not contain any promises, we return it without change.

Otherwise, we lift the `__dict__` and create a promise of a new object of the same class as the input, using `create_object()`. This works fine for what we call *reasonable* objects. Since calling *lift()* is an explicit action, we do not require reasonable objects to be derived from `Reasonable` as we do with serialisation, where such a default behaviour could lead to unexplicable bugs.

noodles.interface.**failed**(*obj*)
Returns True if `obj` is an instance of `Fail`.

class noodles.interface.**PromisedObject**(*workflow*)
Wraps a `Workflow`. The workflow represents the future promise of a Python object. By wrapping the workflow, we can mock the behaviour of this future object and schedule methods that were called by the user as if nothing weird is going on.

class noodles.interface.**Quote**(*promise*)
Quote objects store the contents of a workflow, allowing the workflow to be passed as an argument to a higher

order function without its contents being evaluated. Don't use this object, rather use the functions *quote()* and *unquote()*.

noodles.interface.**quote**(*promise*)
Quote a promise.

noodles.interface.**unquote**(*quoted*)
Unquote a quoted promise.

noodles.interface.**result**(*obj*)
Results are stored on the nodes in the workflow at run time. This function can be used to get at a result of a node in a workflow after run time. This is not a recommended way of getting at results, but can help with debugging.

noodles.interface.**maybe**(*func*)
Calls `f` in a try/except block, returning a *Fail* object if the call fails in any way. If any of the arguments to the call are Fail objects, the call is not attempted.

class noodles.interface.**Fail**(*func*, *fails=None*, *exception=None*)
Signifies a failure in a computation that was wrapped by a `@maybe` decorator. Because Noodles runs all functions from the same context, it is not possible to use Python stack traces to find out where an error happened. In stead we use a `Fail` object to store information about exceptions and the subsequent continuation of the failure.

add_call(*func*)
Add a call to the trace.

is_root_cause
If the field `exception` is set in this object, it means that we are looking at the root cause of the failure.

noodles.interface.**simple_lift**(*obj*)
(scheduled) Create a promise from a plain object.

Runners

class noodles.run.scheduler.**Scheduler**(*verbose=False*, *error_handler=None*, *job_keeper=None*)
Schedules jobs, recieves results, then schedules more jobs as they become ready to compute. This class communicates with a pool of workers by means of coroutines.

run(*connection: noodles.lib.connection.Connection*, *master: noodles.workflow.model.Workflow*)
Run a workflow.

Parameters

- **connection** (Connection) – A connection giving a sink to the job-queue and a source yielding results.
- **master** (Workflow) – The workflow.

noodles.run.hybrid.**hybrid_coroutine_worker**(*selector*, *workers*)
Runs a set of workers, all of them in the main thread. This runner is here for testing purposes.

Parameters

- **selector** (*function*) – A function returning a worker key, given a job.
- **workers** (*dict*) – A dict of workers.

noodles.run.hybrid.**hybrid_threaded_worker**(*selector*, *workers*)
Runs a set of workers, each in a separate thread.

Parameters

- **selector** – A function that takes a hints-tuple and returns a key indexing a worker in the `workers` dictionary.
- **workers** – A dictionary of workers.

Returns A connection for the scheduler.

Return type *Connection*

The hybrid worker dispatches jobs to the different workers based on the information contained in the hints. If no hints were given, the job is run in the main thread.

Dispatching is done in the main thread. Retrieving results is done in a separate thread for each worker. In this design it is assumed that dispatching a job takes little time, while waiting for one to return a result may take a long time.

`noodles.run.hybrid.`**run_hybrid**(*wf*, *selector*, *workers*)

Returns the result of evaluating the workflow; runs through several supplied workers in as many threads.

Parameters

- **wf** (`Workflow` or `PromisedObject`) – Workflow to compute
- **selector** – A function selecting the worker that should be run, given a hint.
- **workers** – A dictionary of workers

Returns result of running the workflow

Serialisation

`noodles.serial.`**pickle**()

Returns a serialisation registry that "just pickles everything".

This registry can be used to bolt-on other registries and keep the pickle as the default. The objects are first pickled to a byte-array, which is subsequently encoded with base64.

`noodles.serial.`**base**()

Returns the Noodles base serialisation registry.

class `noodles.serial.`**Registry**(*parent=None*, *types=None*, *hooks=None*, *hook_fn=None*, *default=None*)

Serialisation registry, keeps a record of *Serialiser* objects.

The Registry keeps a dictionary mapping (qualified) class names to *Serialiser* objects. Given an object, the `__getitem__` method looks for the highest base class that it has a serialiser for. As a fall-back we install a Serialiser matching the Python `object` class.

Detection by object type is not always meaningful or even possible. Before scannning for known base classes the look-up function passes the object through the `hook` function, which should return a string or `None`. If a string is returned that string is used to look-up the serialiser.

Registries can be combined using the '+' operator. The left side argument is than used as `parent` to the new Registry, while the right-hand argument overrides and augments the Serialisers present. The `hook` functions are being chained, such that the right-hand registry takes precedence. The default serialiser is inherrited from the left-hand argument.

decode(*rec*, *deref=False*)

Decode a record to return an object that could be considered equivalent to the original.

The record is not touched if `_noodles` is not an item in the record.

Parameters

- **rec** (*dict*) – A dictionary record to be decoded.
- **deref** (*bool*) – Wether to decode a RefObject. If the encoder wrote files on a remote host, reading this file will be slow and result in an error if the file is not present.

dereference(*data*, *host=None*)
Dereferences RefObjects stuck in the hierarchy. This is a bit of an ugly hack.

encode(*obj*, *host=None*)
Encode an object using the serialisers available in this registry. Objects that have a type that is one of [dict, list, str, int, float, bool, tuple] are send back unchanged.

A host-name can be given as an additional argument to identify the host in the resulting record if the encoder yields any filenames.

This function only treats the object for one layer deep.

Parameters

- **obj** – The object that needs encoding.
- **host** (*str*) – The name of the encoding host.

from_json(*data*, *deref=False*)
Decode the string from JSON to return the original object (if `deref` is true. Uses the `json.loads` function with `self.decode` as object_hook.

Parameters

- **data** (*str*) – JSON encoded string.
- **deref** (*bool*) – Whether to decode records that gave `ref=True` at encoding.

to_json(*obj*, *host=None*, *indent=None*)
Recursively encode `obj` and convert it to a JSON string.

Parameters

- **obj** – Object to encode.
- **host** (*str*) – hostname where this object is being encoded.

class noodles.serial.**Serialiser**(*name='<unknown>'*)
Serialiser base class.

Serialisation classes should derive from *Serialiser* and overload the *encode* and *decode* methods.

Parameters **base** (*type*) – The type that this class is supposed to serialise. This may differ from the type of the object actually being serialised if its class was derived from *base*. The supposed base-class is kept here for reference but serves no immediate purpose.

decode(*cls*, *data*)
Should decode the data to an object of type 'cls'.

Parameters

- **cls** (*type*) – The class is retrieved by the qualified name of the type of the object that was encoded; restored by importing it.
- **data** – The data is the record that was passed to `make_rec` by the encoder.

encode(*obj*, *make_rec*)
Should encode an object of type `self.base` (or derived).

This method receives the object and a function `make_rec`. This function has signature:

```
def make_rec(rec, ref=False, files=None):
    ...
```

If encoding and decoding is somewhat cosuming on resources, the encoder may call with `ref=True`. Then the resulting record won't be decoded until needed by the next job. This is most certainly the case when an external file was written. In this case the filename(s) should be passed as a list by `files=[...]`.

The `files` list is not passed back to the decoder. Rather it is used by noodles to keep track of written files and copy them between hosts if needed. It is the responsibily of the encoder to include the filename information in the passed record as well.

Parameters

- **obj** – Object to be encoded.
- **make_rec** – Function used to pack the encoded data with some meta-data.

class noodles.serial.**SerPath**

decode(*cls*, *data*)

Should decode the data to an object of type 'cls'.

Parameters

- **cls** (*type*) – The class is retrieved by the qualified name of the type of the object that was encoded; restored by importing it.
- **data** – The data is the record that was passed to `make_rec` by the encoder.

encode(*obj*, *make_rec*)

Should encode an object of type `self.base` (or derived).

This method receives the object and a function `make_rec`. This function has signature:

```
def make_rec(rec, ref=False, files=None):
    ...
```

If encoding and decoding is somewhat cosuming on resources, the encoder may call with `ref=True`. Then the resulting record won't be decoded until needed by the next job. This is most certainly the case when an external file was written. In this case the filename(s) should be passed as a list by `files=[...]`.

The `files` list is not passed back to the decoder. Rather it is used by noodles to keep track of written files and copy them between hosts if needed. It is the responsibily of the encoder to include the filename information in the passed record as well.

Parameters

- **obj** – Object to be encoded.
- **make_rec** – Function used to pack the encoded data with some meta-data.

class noodles.serial.**RefObject**(*rec*)

Placeholder object to delay decoding a serialised object until needed by a worker.

class noodles.serial.**AsDict**(*cls*)

decode(*cls*, *data*)

Should decode the data to an object of type 'cls'.

Parameters

- **cls** (*type*) – The class is retrieved by the qualified name of the type of the object that was encoded; restored by importing it.
- **data** – The data is the record that was passed to `make_rec` by the encoder.

encode(*obj*, *make_rec*)

Should encode an object of type `self.base` (or derived).

This method receives the object and a function `make_rec`. This function has signature:

```
def make_rec(rec, ref=False, files=None):
    ...
```

If encoding and decoding is somewhat cosuming on resources, the encoder may call with `ref=True`. Then the resulting record won't be decoded until needed by the next job. This is most certainly the case when an external file was written. In this case the filename(s) should be passed as a list by `files=[...]`.

The `files` list is not passed back to the decoder. Rather it is used by noodles to keep track of written files and copy them between hosts if needed. It is the responsibily of the encoder to include the filename information in the passed record as well.

Parameters

- **obj** – Object to be encoded.
- **make_rec** – Function used to pack the encoded data with some meta-data.

class noodles.serial.**Reasonable**

A Reasonable object is an object which is most reasonably serialised using its `__dict__` property. To deserialise the object, we first create an instance using the `__new__` method, then setting the `__dict__` property manualy. This class is empty, it is used as a tag to designate other objects as reasonable.

class noodles.serial.registry.**RefObject**(*rec*)

Placeholder object to delay decoding a serialised object until needed by a worker.

class noodles.serial.registry.**Registry**(*parent=None*, *types=None*, *hooks=None*, *hook_fn=None*, *default=None*)

Serialisation registry, keeps a record of *Serialiser* objects.

The Registry keeps a dictionary mapping (qualified) class names to *Serialiser* objects. Given an object, the `__getitem__` method looks for the highest base class that it has a serialiser for. As a fall-back we install a Serialiser matching the Python `object` class.

Detection by object type is not always meaningful or even possible. Before scannning for known base classes the look-up function passes the object through the `hook` function, which should return a string or `None`. If a string is returned that string is used to look-up the serialiser.

Registries can be combined using the '+' operator. The left side argument is than used as `parent` to the new Registry, while the right-hand argument overrides and augments the Serialisers present. The `hook` functions are being chained, such that the right-hand registry takes precedence. The default serialiser is inherrited from the left-hand argument.

decode(*rec*, *deref=False*)

Decode a record to return an object that could be considered equivalent to the original.

The record is not touched if `_noodles` is not an item in the record.

Parameters

- **rec** (*dict*) – A dictionary record to be decoded.
- **deref** (*bool*) – Wether to decode a RefObject. If the encoder wrote files on a remote host, reading this file will be slow and result in an error if the file is not present.

dereference(*data*, *host=None*)
Dereferences RefObjects stuck in the hierarchy. This is a bit of an ugly hack.

encode(*obj*, *host=None*)
Encode an object using the serialisers available in this registry. Objects that have a type that is one of [dict, list, str, int, float, bool, tuple] are send back unchanged.

A host-name can be given as an additional argument to identify the host in the resulting record if the encoder yields any filenames.

This function only treats the object for one layer deep.

Parameters

- **obj** – The object that needs encoding.
- **host** (*str*) – The name of the encoding host.

from_json(*data*, *deref=False*)
Decode the string from JSON to return the original object (if `deref` is true. Uses the `json.loads` function with `self.decode` as object_hook.

Parameters

- **data** (*str*) – JSON encoded string.
- **deref** (*bool*) – Whether to decode records that gave `ref=True` at encoding.

to_json(*obj*, *host=None*, *indent=None*)
Recursively encode `obj` and convert it to a JSON string.

Parameters

- **obj** – Object to encode.
- **host** (*str*) – hostname where this object is being encoded.

class noodles.serial.registry.**SerUnknown**(*name='<unknown>'*)

decode(*cls*, *data*)
Should decode the data to an object of type 'cls'.

Parameters

- **cls** (*type*) – The class is retrieved by the qualified name of the type of the object that was encoded; restored by importing it.
- **data** – The data is the record that was passed to `make_rec` by the encoder.

encode(*obj*, *make_rec*)
Should encode an object of type `self.base` (or derived).

This method receives the object and a function `make_rec`. This function has signature:

```
def make_rec(rec, ref=False, files=None):
    ...
```

If encoding and decoding is somewhat cosuming on resources, the encoder may call with `ref=True`. Then the resulting record won't be decoded until needed by the next job. This is most certainly the case when an external file was written. In this case the filename(s) should be passed as a list by `files=[...]`.

The `files` list is not passed back to the decoder. Rather it is used by noodles to keep track of written files and copy them between hosts if needed. It is the responsibily of the encoder to include the filename information in the passed record as well.

Parameters

- **obj** – Object to be encoded.
- **make_rec** – Function used to pack the encoded data with some meta-data.

class noodles.serial.registry.**Serialiser**(*name='<unknown>'*)

Serialiser base class.

Serialisation classes should derive from *Serialiser* and overload the *encode* and *decode* methods.

Parameters **base** (*type*) – The type that this class is supposed to serialise. This may differ from the type of the object actually being serialised if its class was derived from `base`. The supposed base-class is kept here for reference but serves no immediate purpose.

decode(*cls*, *data*)

Should decode the data to an object of type 'cls'.

Parameters

- **cls** (*type*) – The class is retrieved by the qualified name of the type of the object that was encoded; restored by importing it.
- **data** – The data is the record that was passed to `make_rec` by the encoder.

encode(*obj*, *make_rec*)

Should encode an object of type `self.base` (or derived).

This method receives the object and a function `make_rec`. This function has signature:

```
def make_rec(rec, ref=False, files=None):
    ...
```

If encoding and decoding is somewhat cosuming on resources, the encoder may call with `ref=True`. Then the resulting record won't be decoded until needed by the next job. This is most certainly the case when an external file was written. In this case the filename(s) should be passed as a list by `files=[...]`.

The `files` list is not passed back to the decoder. Rather it is used by noodles to keep track of written files and copy them between hosts if needed. It is the responsibily of the encoder to include the filename information in the passed record as well.

Parameters

- **obj** – Object to be encoded.
- **make_rec** – Function used to pack the encoded data with some meta-data.

Worker executable

Streams

Coroutine streaming module

Note: In a break with tradition, some classes in this module have lower case names because they tend to be used as function decorators.

We use coroutines to communicate messages between different components in the Noodles runtime. Coroutines can have input or output in two ways *passive* and *active*. An example:

```
def f_pulls(coroutine):
    for msg in coroutine:
        print(msg)

def g_produces(lines):
    for l in lines:
        yield lines

lines = ['aap', 'noot', 'mies']

f_pulls(g_produces(lines))
```

This prints the words 'aap', 'noot' and 'mies'. This same program could be written where the co-routine is the one receiving messages:

```
def f_receives():
    while True:
        msg = yield
        print(msg)

def g_pushes(coroutine, lines):
    for l in lines:
        coroutine.send(l)

sink = f_receives()
sink.send(None)  # the co-routine needs to be initialised
                 # alternatively, .next() does the same as .send(None)
g_pushes(sink, lines)
```

The action of creating a coroutine and setting it to the first `yield` statement can be performed by a little decorator:

```
from functools import wraps

def coroutine(f):
    @wraps(f)
    def g(*args, **kwargs):
        sink = f(*args, **kwargs)
        sink.send(None)
        return sink

    return g
```

Pull and push

The *pull* and *push* classes capture the idea of pushing and pulling coroutines, wrapping them in an object. These objects can then be chained using the >> operator. Example:

```
>>> from noodles.lib import (pull_map, pull_from)
>>> @pull_map
... def square(x):
...     return x*x
...
>>> squares = pull_from(range(10)) >> square
>>> list(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Queues

Queues in python are thread-safe objects. We can define a new *Queue* object that uses the python `queue.Queue` to buffer and distribute messages over several threads:

```
import queue

class Queue(object):
    def __init__(self):
        self._q = queue.Queue()

    def source(self):
        while True:
            msg = self._q.get()
            yield msg
            self._q.task_done()

    @coroutine
    def sink(self):
        while True:
            msg = yield
            self._q.put(msg)

    def wait(self):
        self.Q.join()
```

Note, that both ends of the queue are, as we call it, passive. We could make an active source (it would become a normal function), taking a call-back as an argument. However, we're designing the Noodles runtime so that it easy to interleave functionality. Moreover, the *Queue* object is only concerned with the state of its own queue. The outside universe is only represented by the `yield` statements, thus preserving the principle of encapsulation.

`noodles.lib.`**`decorator`**(*f*)
: Creates a paramatric decorator from a function. The resulting decorator will optionally take keyword arguments.

`noodles.lib.`**`coroutine`**(*f*)
: A sink should be send `None` first, so that the coroutine arrives at the `yield` position. This wrapper takes care that this is done automatically when the coroutine is started.

`class` `noodles.lib.`**`stream`**
: Base class for *pull* and *push* coroutines.

`class` `noodles.lib.`**`pull`**(*fn*)
: A *pull* coroutine pulls from a source, yielding values. *pull* Objects can be chained using the >> operator.

  A *pull* object acts as a function of one argument, being the source that the coroutine will pull from. This source argument must always be a thunk (function of zero arguments) returning an iterable.

`class` `noodles.lib.`**`push`**(*fn*, *dont_wrap=False*)
: A *push* coroutine pushes to a sink, receiving values through `yield` statements. *push* Objects can be chained using the >> operator.

  A *push* object acts as a function of one argument, being the sink that the coroutine will send to. This sink argument must always be a thunk (function of zero arguments) returning a coroutine.

`class` `noodles.lib.`**`pull_map`**(*f*)
: A *pull_map* decorates a function of a single argument, to become a *pull* object. The resulting *pull* object pulls object from a source yielding values mapped through the given function.

  This is equivalent to:

```
@pull
def g(source):
    yield from map(f, source())
```

where `f` is the function being decorated.

The >> operator on this class is optimised such that only a single loop will be created when chained with another *pull_map*.

Also, a *pull_map* may be chained to a function directly, including the given function in the loop.

class noodles.lib.**push_map**(*f*)

A *push_map* decorates a function of a single argument, to become a *push* object. The resulting *push* object receives values through `yield` and sends them on after mapping through the given function.

This is equivalent to:

```
@push
def g(sink):
    sink = sink()
    while True:
        x = yield
        sink.send(f(x))
```

where `f` is the function being decorated.

The >> operator on this class is optimised such that only a single loop will be created when chained with another *push_map*.

Also, a *push_map* may be chained to a function directly, including the given function in the loop.

noodles.lib.**sink_map**(*f*)

The *sink_map()* decorator creates a *push* object from a function that returns no values. The resulting sink can only be used as an end point of a chain.

Equivalent code:

```
@push
def sink():
    while True:
        x = yield
        f(x)
```

noodles.lib.**broadcast**(**sinks_*)

The *broadcast()* decorator creates a *push* object that receives a message by `yield` and then sends this message on to all the given sinks.

noodles.lib.**branch**(**sinks_*)

The *branch()* decorator creates a *pull* object that pulls from a single source and then sends to all the sinks given. After all the sinks received the message, it is yielded.

noodles.lib.**patch**(*source*, *sink*)

Create a direct link between a source and a sink.

Implementation:

```
sink = sink()
for value in source():
    sink.send(value)
```

noodles.lib.**pull_from**(*iterable*)
Creates a *pull* object from an iterable.

Parameters **iterable** (Iterable) – an iterable object.

Return type *pull*

Equivalent to:

```
pull(lambda: iter(iterable))
```

noodles.lib.**push_from**(*iterable*)
Creates a *push* object from an iterable. The resulting function is not a coroutine, but can be chained to another *push*.

Parameters **iterable** (Iterable) – an iterable object.

Return type *push*

class noodles.lib.**Connection**(*source*, *sink*, *aux=None*)
Combine a source and a sink. These should represent the IO of some object, probably a worker. In this case the source is a coroutine generating results, while the sink needs to be fed jobs.

setup()
Activate the source and sink functions and return them in that order.

Returns source, sink

Return type tuple

class noodles.lib.**Queue**(*end_of_queue=<class 'noodles.lib.queue.EndOfQueue'>*)
A *Queue* object hides a queue.Queue object behind a source and sink interface.

sink
Receives items that are put on the queue. Pushing the end-of-queue message through the sink will put it on the queue, and will also result in a StopIteration exception being raised.

source
Pull items from the queue. When end-of-queue is encountered the generator returns after re-inserting the end-of-queue message on the queue for other sources to pick up. This way, if many threads are pulling from this queue, they all get the end-of-queue message.

close()
Sends end_of_queue message to the queue. Doesn't stop running sinks.

flush()
Erases queue and set end-of-queue message.

noodles.lib.**thread_pool**(**workers*, *results=None*, *end_of_queue=<class 'noodles.lib.queue.EndOfQueue'>*)
Returns a *pull* object, call it r, starting a thread for each given worker. Each thread pulls from the source that r is connected to, and the returned results are pushed to a *Queue*. r yields from the other end of the same *Queue*.

The target function for each thread is *patch()*, which can be stopped by exhausting the source.

If all threads have ended, the result queue receives end-of-queue.

Parameters

- **results** (*Connection*) – If results should go somewhere else than a newly constructed *Queue*, a different *Connection* object can be given.

- **end_of_queue** – end-of-queue signal object passed on to the creation of the *Queue* object.

Return type *pull*

noodles.lib.**thread_counter**(*finalize*)
Modifies a thread target function, such that the number of active threads is counted. If the count reaches zero, a finalizer is called.

noodles.lib.**object_name**(*obj*)
Get the qualified name of an object. This will obtain both the module name from __module__ and object name from __name__, and concatenate those with a '.'. Examples:

```
>>> from math import sin
>>> object_name(sin)
'math.sin'
```

```
>>> def f(x):
...     return x*x
...
>>> object_name(f)
'__main__.f'
```

To have a qualified name, an object must be defined as a class or function in a module (__main__ is also a module). A normal instantiated object does not have a qualified name, even if it is defined and importable from a module. Calling *object_name()* on such an object will raise AttributeError.

noodles.lib.**look_up**(*name*)
Obtain an object from a qualified name. Example:

```
>>> look_up('math.sin')
<built-in function sin>
```

This function should be considered the reverse of *object_name()*.

noodles.lib.**importable**(*obj*)
Check if an object can be serialised as a qualified name. This is done by checking that a look_up(object_name(obj)) gives back the same object.

noodles.lib.**deep_map**(*f*, *root*)
Sibling to *inverse_deep_map()*. As map() maps over an iterable, *deep_map()* maps over a structure of nested ''dict''s and ''list''s. Every object is passed through f recursively. That is, first root is mapped, next any object contained in its result, and so on.

No distinction is made between tuples and lists. This function was created with encoding to JSON compatible data in mind.

noodles.lib.**inverse_deep_map**(*f*, *root*)
Sibling to *deep_map()*. Recursively maps objects in a nested structure of list and dict objects. Where *deep_map()* starts at the top, *inverse_deep_map()* starts at the bottom. First, if root is a list or dict, its contents are linverse_deep_mapled. Then at the end, the entire object is passed through f.

This function was created with decoding from JSON compatible data in mind.

noodles.lib.**unwrap**(*f*)
Safely obtain the inner function of a previously wrapped (or decorated) function. This either returns f.__wrapped__ or just f if the latter fails.

noodles.lib.**is_unwrapped**(*f*)
If f was imported and then unwrapped, this function might return True.

4.4.2 The Noodles Scheduler

The Noodles scheduler is completely separated from the worker infrastructure. The scheduler accepts a single worker as an argument. This worker provides the scheduler with two coroutines. One acts as a generator of results, the other as a sink for jobs (the scheduler calls the `send()` method on it).

Both jobs and results are accompanied by a unique key to identify the associated job. The scheduler loops over the results as follows (more or less):

```
for (key, result) in source:
    """process result"""
    ...

    for node in workflow.nodes:
        if node.ready():
            sink.send((node.key, node.job))
```

Local workers

The single worker

It is the responsibility of the worker to keep a queue where so desired. A single result may trigger many new nodes to be ready for evaluation. This means that either the jobs or the results must be buffered in a queue. In the simplest case we have a single worker in the same thread as the scheduler.

Fig. 3: Sequence diagram for a single threaded execution model.

The worker code looks like this:

```
from noodles.coroutines import (IOQueue, Connection)
from noodles.run_common import run_job

def single_worker():
    """Sets up a single worker co-routine."""
    jobs = IOQueue()

    def get_result():
        source = jobs.source()

        for key, job in source:
            yield (key, run_job(job))

    return Connection(get_result, jobs.sink)
```

The `IOQueue` class wraps a standard Python queue. It provides a `sink` member pushing elements onto the queue, and a `source` member yielding elements from the queue, calling `Queue.task_done()` when the coroutine regains control. The `Connection` class packs a coroutine source (a generator) and a sink. Together these objects provide a plug-board interface for the scheduler and a hierarchy of workers.

Now, when the scheduler calls `sink.send(...)`, the job is pushed onto the queue that is created in `single_worker()`. When the scheduler iterates over the results, `get_result()` feeds it results that it computes itself (through `run_job`).

The Python queue is thread-safe. We may call `jobs.source()` in a different thread in another worker. This worker then safely pulls jobs from the same queue.

The Threaded worker

To have several workers run in tandem we need to keep a result queue in addition to the job queue. In the next sequence diagram we see how any number of threads are completely decoupled from the thread that manages the scheduling.

Fig. 4: Sequence diagram where the actual job execution is deferred to one or more additional threads.

In Python source this looks as follows:

```
def threaded_worker(n_threads):
    """Sets up a number of threads, each polling for jobs."""
    job_q = IOQueue()
    result_q = IOQueue()

    worker_connection = QueueConnection(job_q, result_q)
    scheduler_connection = QueueConnection(result_q, job_q)

    def worker(source, sink):
        for key, job in source:
            sink.send((key, run_job(job)))

    for i in range(n_threads):
        t = threading.Thread(
            target=worker,
            args=worker_connection.setup())

        t.daemon = True
        t.start()

    return scheduler_connection
```

The Hybrid worker

Fig. 5: Sequence diagram where the jobs get dispatched, each to a worker selected by a dispatcher.

Remote workers

Xenon

Fireworks

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W