Python-based Frameworks for KM3NeT

Tamás Gál from ECAP/Erlangen, PyGamma19 Conference, March 2019, Heidelberg

github.com/tamasgal (github.com/tamasgal) / tamas.gal@fau.de (mailto:tamas.gal@fau.de)

This is an interactive presentation using Jupyter and the RISE plugin. This PDF was created to provide a static version of the content and includes some additional notes which are hidden during the talk.

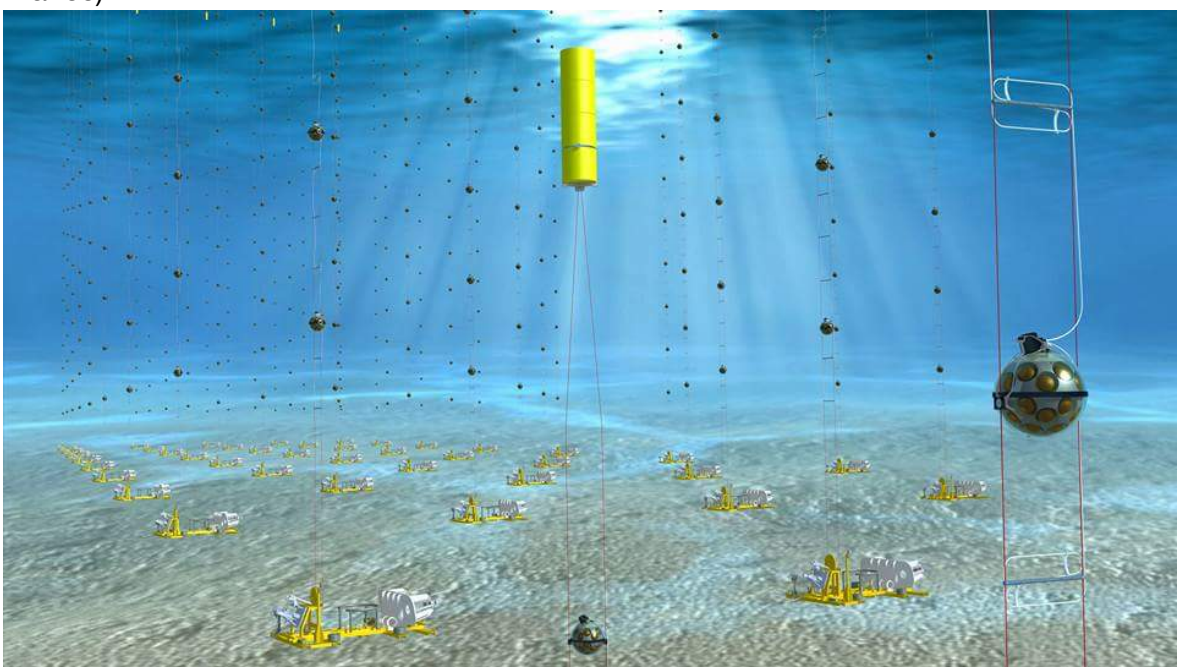
In [1]:
```
from IPython.core.magic import register_line_magic

@register_line_magic
def shorterr(line):
    """Show only the exception message if one is raised."""
    try:
        output = eval(line)
    except Exception as e:
        print("\x1b[31m\x1b[1m{e.__class__.__name__}: {e}\x1b[0m".form
    else:
        return output

del shorterr
```

Purpose of this Talk

- inspire other people
- get inspired by other people
- collaborate with each other

What is KM3NeT?

- a (multi) cubic kilometer (**KM3**) **Ne**utrino **T**elescope in the Mediterranean: https://www.km3net.org (https://www.km3net.org)
- currently under construction but already operating at two different sites (Italy and France)



The Frameworks

- **aanet**
  - low level analysis framework with high level capabilities
  - main dev: Aart Heijboer - Nikhef / Amsterdam

- **OrcaNet** – https://git.km3net.de/ml/orcanet (https://git.km3net.de/ml/orcanet)
  - deep learning training organiser
  - main devs: Michael Moser, Stefan Reck - ECAP / Erlangen
  - docs: https://ml.pages.km3net.de/OrcaNet (https://ml.pages.km3net.de/OrcaNet)

- **KM3Pipe** – https://git.km3net.de/km3py/km3pipe (https://git.km3net.de/km3py/km3pipe)
  - multi-purpose framework
  - main devs: Tamás Gál, Johannes Schumann - ECAP / Erlangen
  - credits: Moritz Lotze
  - docs: https://km3pipe.rtfd.io (https://km3pipe.rtfd.io)

aanet

Low Level Analysis Framework for KM3NeT (closed source)

- main dev: Aart Heijboer - Nikhef / Amsterdam
- tailored to KM3NeT specific dataformats and workflows
- mostly written in C++
- based on ROOT (https://root.cern.ch) structures
- utilises PyROOT (https://root.cern.ch/pyroot) for high level access
- the current alpha release connects ROOT (https://root.cern.ch) with NumPy (https://www.numpy.org) (the reason I mention this framework)

ROOT Introspection to generate NumPy `dtype`s

The dtype for every aanet (or ROOT) class can be automatically generated from the ROOT dictionary information.

```
>>> import ROOT
>>> import aa
>>> aa.aanumpy.get_dtype( Hit )
dtype({'names':['id','dom_id','channel_id','tdc','tot','trig'
,'pmt_id','t','a','pos','dir','pure_t','pure_a','type','origi
n','pattern_flags'], 'formats':['<i4','<i4','<u4','<u4','<u4'
,'<i4','<i4','<f8','<f8',[('x', '<f8'), ('y', '<f8'), ('z', '
<f8')],[('x', '<f8'), ('y', '<f8'), ('z', '<f8')],'<f8','<f8'
,'<i4','<i4','<u4'], 'offsets':[64,68,72,76,80,84,88,96,104,1
12,136,160,168,176,180,184], 'itemsize':192})
```

And a vector of Hits becomes a (structured) numpy array…

Rough Sketch of the ROOT to numpy conversion (1/3)

```
nptypes = {
    "int": np.int32,
    "long int": np.int64,
    "unsigned int": np.uint32,
    "unsigned long": np.uint64,
    "double": np.float64,
    "size_t": np.uint32
}
```

Rough Sketch of the ROOT to numpy conversion (2/3)

In [2]:
```
def get_dtype(rootclass):
    D = collections.defaultdict( list )
    L = rootclass.GetListOfDataMembers()
    for m in L:
        type_name, name, u_size = m.GetTypeName(), m.GetName(), m.GetU
        if m.IsBasic():
            dtype = nptypes[type_name]
        else:
            value_type_np = nptypes[type_name.split("<",1)[1].rsplit('
            dtype = np.dtype((np.uint8, m.GetUnitSize()))
        D['names'].append(name)
        D['formats'].append(dtype)
        D['offset'].append(m.GetOffset())
    D['aligned'] = False
    D['itemsize'] = rootclass.Size()
    return np.dtype(D)
```

Rough Sketch of the ROOT to numpy conversion (3/3)

In [3]:
```
def numpyfi(root_stl_vector):
    dtype = get_dtype(root_stl_vector[0].Class())
    buf = ROOT._avoid(root_stl_vector[0])
    buf.SetSize(dtype.itemsize * len(root_stl_vector))
    return np.recarray((len(root_stl_vector),), dtype=dtype, buf=buf)
```

Actual User API

```
hits = aa.numpyfi(evt.hits)  # evt.hits is a std::Vector<Hit>
...
hits.time  # a numpy array of the hit times
hits.pos.x  # a numpy array of x-coordinates of the hits
```

Note: no copy of data, just the creation of the container structure!

OrcaNet

Deep Learning Training Organiser by Michael Moser and Stefan Reck – ECAP / Erlangen

https://ml.pages.km3net.de/OrcaNet (https://ml.pages.km3net.de/OrcaNet)

- Takes care of training, validating, logging, plotting and continuation of previous traninings

```
from orcanet.core import Organizer

input_definition = "configuration.toml"
output_folder = "path/to/output"

model = ... # compiled keras model

organizer = Organizer(output_folder, input_definition)
organizer.train_and_validate(model)
```

Input Definition

- uses a TOML (https://en.wikipedia.org/wiki/TOML) based file format
- easy support for multiple files and multiple inputs

```
[input xy]
train_files = ['data/xy_train_file_1.h5', 'data/xy_train_file
_1.h5']
train_files = ['data/xy_val_file_1.h5', 'data/xy_val_file_1.h
5']

[input yz]
train_files = ['data/yz_train_file_1.h5', 'data/yz_train_file
_1.h5']
train_files = ['data/yz_val_file_1.h5', 'data/yz_val_file_1.h
5']
```

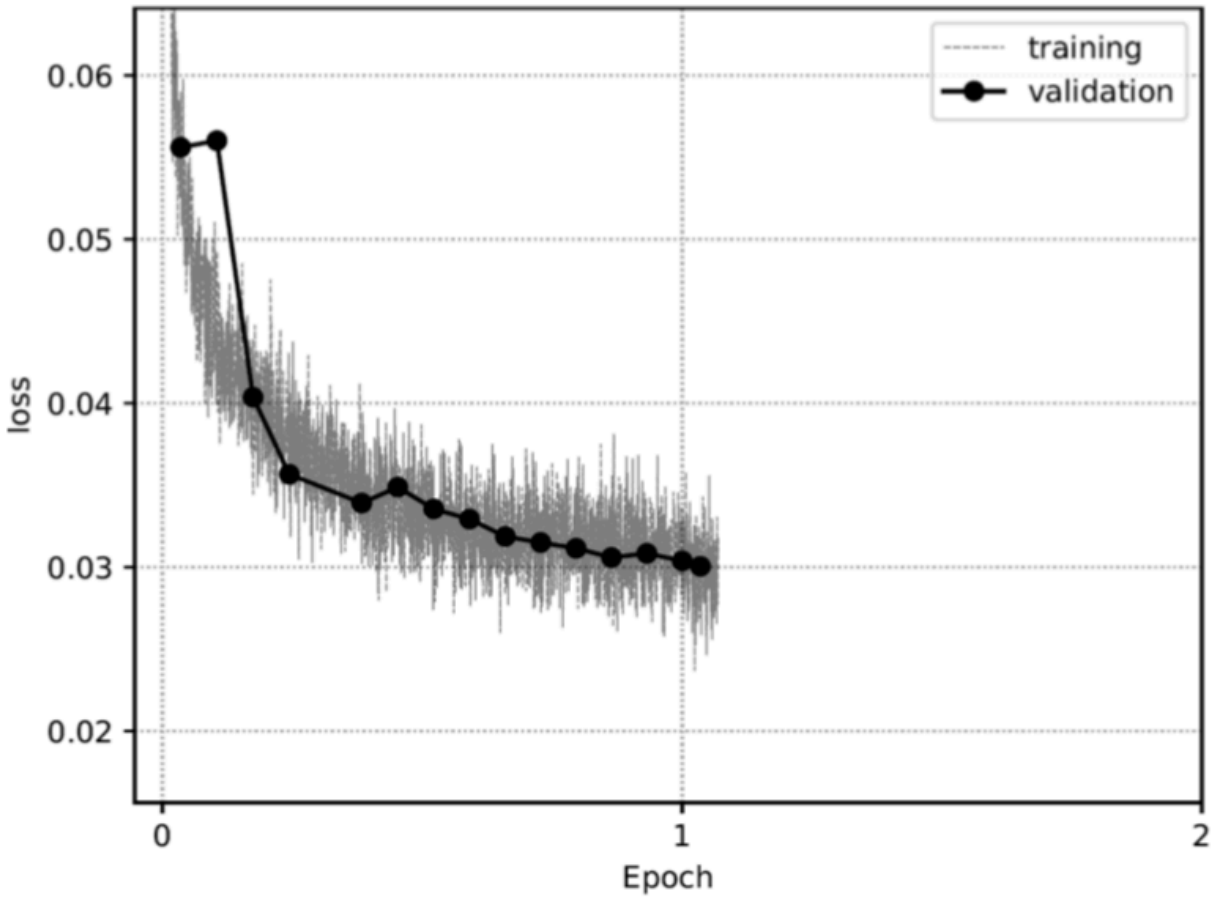
Output Summary

- logged in human-readable format
- also available as numpy structured array via `organizer.get_summary_data()`

Example (`output_folder/summary.txt`)

```
Epoch    | LR       | train_loss | val_loss | train_acc | val_
acc
--------+----------+------------+----------+-----------+-----
---
0.03488 | 0.005    | 0.07776    | 0.05562  | 0.971     | 0.98
08
0.06971 | 0.00465  | 0.05034    | nan      | 0.9822    | nan
0.1045  | 0.004324 | 0.04542    | 0.05603  | 0.9841    | 0.98
04
0.1394  | 0.004022 | 0.04231    | nan      | 0.9853    | nan
...
```

Example (`output_folder/summary.pdf`)



KM3Pipe

Multipurpose Pipeline Framework

https://km3pipe.rtfd.io (https://km3pipe.rtfd.io)

- via **pip**: `pip install km3pipe`
- or using Docker (https://docker.com): `docker run -it git.km3net.de/km3pipe`
- to run the test suite: `km3pipe test`

In [4]:
```
import km3pipe as kp
import km3modules as km  # a collection of commonly used modules
import numpy as np

kp.version
```

Out[4]: '8.12.2'

History of KM3Pipe (1/2)

- **2012**: started as a small project to unify the work with different fileformats and frameworks used for ANTARES and KM3NeT
- main focus on Python based scientific libraries (NumPy (https://numpy.org) and SciPy (https://SciPy.org))
- **2013**: HDF5 (https://www.hdfgroup.org) is used as standard dataformat for output (the input was all kinds of ROOT, ASCII and custom binary formats)
- **2014**: renamed to KM3Pipe and a fresh repository has been created to consolidate previous works
- **2015**: official framework for KM3NeT realtime detector monitoring
  - Moritz Lotze joined me!
- acknowledged "alternative" framework for data analysis in KM3NeT
- **2016**: the HDF5 format layout was defined and officially introduced
- **2018**: Moritz left the collaboration :(but Johannes Schumann joined (thanks!)

History of KM3Pipe (2/2)

- went over multiple design overhauls during the years to improve performance
  - a lot of cython, ctypes, and C++ wrapper experiments
- we always try to follow best practices:
  - aim for good test coverage (currently overall 72%, core&dataclasses: 95%)
    - https://km3py.pages.km3net.de/km3pipe/coverage/ (https://km3py.pages.km3net.de/km3pipe/coverage/)
  - continuously integrated from the very beginning (Jenkins, Travis, then self-hosted GitLab+Docker)
  - try to stick to Python STL and scientific libs (numpy/pytables)
  - YAGNI, DRY, KISS...
- in **2018** we tried to drop Python 2 support but eventually gave up due to dependencies on other software not supporting Python 3
- today the team consists of three active commiters (13 unique commiters in total)

The Pipeline Concept

- overall concept roughly inspired by IceTray (https://github.com/claudiok/icetray-icetray), a C++/Python based hybrid framework developed by the IceCube Neutrino Observatory (https://icecube.wisc.edu)
- three main ingrediences: `Pipeline`, `Module` and `Blob`
- the `Pipeline` creates the main structure which holds everything together
- everything else attached to it is a `Module`, either a
  - a dead simple Python function
  - or an actual `km3pipe.Module`
- the data is passed from module to module using a `Blob`:
  - started as `Blob = dict`
  - then after a while transitioned to `class Blob(dict): ...` with a nicer `__str__`
  - currently `class Blob(OrderedDict): ...` with an even nicer `__str__` and `__getitem__`

Why a Pipeline?

- people tend to collaborate easier when there is some kind of code structure
- a `Pipeline` gives an uncluttered overview of the main design
- `Module`s are easy to understand and lightweight
- exchangable blocks of code

Modularity Example (1/2)

A pipeline for an awesome neutrino reconstruction using a MC file on disk:

```
pipe = Pipeline()
pipe.attach(kp.io.AanetPump, filename="some_mc_file.root")
pipe.attach(MyAwesomeEventFilter)
pipe.attach(YourAwesomeNeutrinoReco)
pipe.attach(kp.io.HDF5Sink, filename="reco_results.h5")
```

Modularity Example (2/2)

The same pipeline which reconstructs live events from the detector:

```
pipe = Pipeline()
pipe.attach(kp.io.CHPump, host='1.23.5.42', port=50023)
pipe.attach(MyAwesomeEventFilter)
pipe.attach(YourAwesomeNeutrinoReco)
pipe.attach(kp.io.HDF5Sink, filename="reco_results.h5")
```

A Basic `Pipeline` Example (Python function as `Module`)

In [5]:
```
def a_module(blob):
    """A module, represented by a vanilla Python function"""
    print("helo")
    return blob
```

In [6]:
```
pipe = kp.Pipeline()
```

In [7]:
```
pipe.attach(a_module)
```

In [8]:
```
pipe.drain(3);  # drain 3 cycles
```

```
Pipeline and module initialisation took 0.013s (CPU 0.007s).
helo
helo
helo
============================================================
3 cycles drained in 0.012914s (CPU 0.007001s). Memory peak: 148.91 M
B
  wall  mean: 0.000049s  medi: 0.000044s  min: 0.000026s  max: 0.000
078s  std: 0.000022s
  CPU   mean: 0.000084s  medi: 0.000052s  min: 0.000026s  max: 0.000
176s  std: 0.000066s
```

A Basic `Pipeline` Example (using `km3pipe.Module`)

In [9]:
```
class AModule(kp.Module):
    """A KM3Pipe Module"""
    def configure(self):  # called once before "draining"
        print("Configuring myself...")
        self.foo = 1

    def process(self, blob):  # called in each cycle
        print(f"Foo = {self.foo}")
        self.foo += 1
        return blob

    def finish(self):  # called when the pipeline finished "draining"
        print("I am done...")
```

In [10]:
```
pipe = kp.Pipeline()
pipe.attach(AModule)
pipe.drain(3);
```

```
Configuring myself...
Pipeline and module initialisation took 0.000s (CPU 0.000s).
Foo = 1
Foo = 2
Foo = 3
I am done...
============================================================
3 cycles drained in 0.000455s (CPU 0.000593s). Memory peak: 149.16 M
B
  wall  mean: 0.000031s  medi: 0.000031s  min: 0.000022s  max: 0.000
041s  std: 0.000008s
  CPU   mean: 0.000046s  medi: 0.000034s  min: 0.000028s  max: 0.000
076s  std: 0.000021s
```

The `Blob`

In [11]:
```
blob = kp.Blob({'a': 1, 'b': 'foo'})
print(blob)
```

```
Blob (2 entries):
 'a' => 1
 'b' => 'foo'
```

In [12]:
```
isinstance(blob, dict)
```

Out[12]:
```
True
```

In [13]:
```
%shorterr blob['c']
```

```
ERROR ++ Blob: No key named 'c' found in Blob.
Available keys: a, b

KeyError: 'c'
```

The `Blob` is Passed Around

A typical pipeline consists of a `Blob` generator (aka `Pump`), which generates/populates the `Blob`. During a cycle, the `Blob` is passed from `Module` to `Module` and each one is able to modify it.

In [14]:
```
def a_module(blob):
    blob['random_number'] = np.random.rand()
    return blob

def b_module(blob):
    print(blob['random_number'])
    return blob

pipe = kp.Pipeline()
pipe.attach(a_module)
pipe.attach(b_module)
pipe.drain(3)
```

```
Pipeline and module initialisation took 0.000s (CPU 0.000s).
0.2450668729317248
0.9406669610323172
0.387145480652087
============================================================
3 cycles drained in 0.000472s (CPU 0.000659s). Memory peak: 149.16 M
B
  wall  mean: 0.000085s  medi: 0.000096s  min: 0.000045s  max: 0.000
114s  std: 0.000029s
  CPU   mean: 0.000130s  medi: 0.000146s  min: 0.000079s  max: 0.000
165s  std: 0.000036s
```

Out[14]:
```
Blob()
```

Basic Control Flow: `every`

In [15]:
```
class StatusBar(kp.Module):
    def configure(self):
        self.index = 0

    def process(self, blob):
        print(f"---- {self.index * self.every} ----")
        self.index += 1
        return blob

pipe = kp.Pipeline()
pipe.attach(StatusBar, every=50)
pipe.drain(150);
```

```
Pipeline and module initialisation took 0.000s (CPU 0.000s).
---- 0 ----
---- 50 ----
---- 100 ----
============================================================
150 cycles drained in 0.001076s (CPU 0.001260s). Memory peak: 149.16
MB
  wall  mean: 0.000004s  medi: 0.000004s  min: 0.000004s  max: 0.000
023s  std: 0.000002s
  CPU   mean: 0.000005s  medi: 0.000004s  min: 0.000004s  max: 0.000
052s  std: 0.000005s
```

Basic Control Flow: `only_if`

In [16]:
```
def a_module(blob):
    x = np.random.rand()
    if x > 0.8:
        blob['random_number'] = x
    return blob

def b_module(blob):
    print(blob['random_number'])
    return blob

pipe = kp.Pipeline()
pipe.attach(a_module)
pipe.attach(b_module, only_if='random_number')
pipe.drain(10);
```

```
Pipeline and module initialisation took 0.000s (CPU 0.000s).
0.9119095529823243
0.8378649853576591
============================================================
10 cycles drained in 0.000366s (CPU 0.000482s). Memory peak: 149.16
MB
  wall  mean: 0.000017s  medi: 0.000012s  min: 0.000009s  max: 0.000
030s  std: 0.000008s
  CPU   mean: 0.000023s  medi: 0.000014s  min: 0.000012s  max: 0.000
058s  std: 0.000015s
```

Configuring Module Parameters

Two types of parameters:

- optional (with default values)
- required

Configured inside the `configure()` method using `self.get()` and `self.require()`

Configuring Module Parameters - Optional Parameters (1/2)

In [17]:
```
class AModule(kp.Module):
    def configure(self):
        self.foo = self.get('foo', default=23)  # if default is not se

    def process(self, blob):
        print(f"foo = {self.foo}")
```

In [18]:
```
pipe = kp.Pipeline()
pipe.attach(AModule)
pipe.drain(3);
```

```
Pipeline and module initialisation took 0.000s (CPU 0.000s).
foo = 23
foo = 23
foo = 23
============================================================
3 cycles drained in 0.000252s (CPU 0.000398s). Memory peak: 149.16 M
B
  wall  mean: 0.000022s  medi: 0.000019s  min: 0.000018s  max: 0.000
028s  std: 0.000004s
  CPU   mean: 0.000041s  medi: 0.000028s  min: 0.000018s  max: 0.000
076s  std: 0.000025s
```

Configuring Module Parameters - Optional Parameters (2/2)

In [19]:
```
pipe = kp.Pipeline()
pipe.attach(AModule, foo=5)
pipe.drain(3);
```

```
Pipeline and module initialisation took 0.000s (CPU 0.000s).
foo = 5
foo = 5
foo = 5
============================================================
3 cycles drained in 0.000273s (CPU 0.000351s). Memory peak: 149.16 M
B
  wall  mean: 0.000027s  medi: 0.000028s  min: 0.000021s  max: 0.000
031s  std: 0.000004s
  CPU   mean: 0.000050s  medi: 0.000031s  min: 0.000026s  max: 0.000
091s  std: 0.000030s
```

Configuring Module Parameters - Required Parameters

In [20]:
```
class AModule(kp.Module):
    def configure(self):
        self.foo = self.require('foo')

    def process(self, blob):
        print(f"foo = {self.foo}")
```

In [21]:
```
pipe = kp.Pipeline()
%shorterr pipe.attach(AModule)
```

```
TypeError: <class '__main__.AModule'> requires the parameter 'foo'.
```

Logging

- KM3Pipe uses the `logging` module
- adds some eye candy (colours, symbols...)
- hierarchical (dot separation)
- integrates them into the `Module` s: `self.log`
- two additional log levels (currently 45 and 46)
  - `log.once()` (executed only once, determined by position in source-code)
  - `log.deprecation()`

In [22]:
```
log_a = kp.logger.get_logger('log_a')
log_b = kp.logger.get_logger('log_b')
log_c = kp.logger.get_logger('log_c')

log_a.setLevel('DEBUG')
log_b.setLevel('INFO')
```

In [23]:
```
log_a.debug("Entering the debug log function...")
log_b.info("Preparing some logging demo.")
log_c.warning("The next line will be an error!")
log_a.error("It starts to get critical.")
log_b.critical("FAIL")
log_a.once("I show this only once!", identifier=5)  # identifier neede
log_b.deprecation("That's so 90es, try the new hipster API!")
```

```
DEBUG ++ log_a: Entering the debug log function...
INFO ++ log_b: Preparing some logging demo.
WARNING ++ log_c: The next line will be an error!
ERROR ++ log_a: It starts to get critical.
CRITICAL ++ log_b: FAIL
ONCE ++ log_a: I show this only once!
DEPRECATION ++ log_b: That's so 90es, try the new hipster API!
```

The `printer`

In [24]:
```
printer = kp.logger.get_printer('log_a')
```

In [25]:
```
printer("Just a message")
```

```
++ log_a: Just a message
```

In [26]:
```
log_a.warning("Same symbol colour...")
```

```
WARNING ++ log_a: Same symbol colour...
```

Printing and Logging in a `Module`

The logger and the printer are initialised upon `Module` creation and acccessible via `self.log` and `self.print`.

In [27]:
```
class AModule(kp.Module):
    def configure(self):
        self.log.info("configure called")
    def process(self, blob):
        self.print("processing blob!")
    def finish(self):
        self.print("Finishing")
```

In [28]:
```
kp.logger.get_logger('AModule').setLevel('INFO')

pipe = kp.Pipeline()
pipe.attach(AModule)
pipe.drain(2);
```

```
INFO ++ AModule: configure called

Pipeline and module initialisation took 0.001s (CPU 0.001s).
++ AModule: processing blob!
++ AModule: processing blob!
++ AModule: Finishing
============================================================
2 cycles drained in 0.000617s (CPU 0.000839s). Memory peak: 149.44 M
B
  wall  mean: 0.000023s  medi: 0.000023s  min: 0.000020s  max: 0.000
027s  std: 0.000004s
  CPU   mean: 0.000029s  medi: 0.000029s  min: 0.000025s  max: 0.000
033s  std: 0.000004s
```

Services: Expose Functionality

In [29]:
```
class RandomService(kp.Module):
    def configure(self):
        seed = self.get('seed', default=23)
        self.random_state = np.random.RandomState(seed)
        self.expose(self.uniform, 'uniform')
        self.expose(self.rand, 'rand')
    def uniform(self): return self.random_state.uniform()
    def rand(self): return self.random_state.rand()

class AModule(kp.Module):
    def process(self, blob):
        print(self.services['uniform'](), self.services['rand']())
        return blob
```

In [30]:
```
pipe = kp.Pipeline()
pipe.attach(RandomService)
pipe.attach(AModule)
pipe.drain(2);
```

```
Pipeline and module initialisation took 0.000s (CPU 0.000s).
0.5172978838465893 0.9469626038148141
0.7654597593969069 0.2823958439671127
============================================================
2 cycles drained in 0.000408s (CPU 0.000519s). Memory peak: 149.44 M
B
  wall  mean: 0.000052s  medi: 0.000052s  min: 0.000041s  max: 0.000
063s  std: 0.000011s
  CPU   mean: 0.000067s  medi: 0.000067s  min: 0.000056s  max: 0.000
079s  std: 0.000012s
```

Performance Statistics

Built-in tracking of CPU/wall times and peak memory usage.

In [31]:
```
import time

def bottleneck(blob):
    time.sleep(1)
    return blob

pipe = kp.Pipeline(timeit=True)
pipe.attach(RandomService)
pipe.attach(AModule)
pipe.attach(bottleneck)
pipe.drain(3);
```

```
Pipeline and module initialisation took 0.000s (CPU 0.000s).
0.5172978838465893 0.9469626038148141
0.7654597593969069 0.2823958439671127
0.22104536326165258 0.6862220852374669
============================================================
3 cycles drained in 3.005004s (CPU 0.011713s). Memory peak: 149.44 M
B
  wall  mean: 1.001527s  medi: 1.001481s  min: 1.001232s  max: 1.001
867s  std: 0.000261s
  CPU   mean: 0.003735s  medi: 0.002872s  min: 0.002831s  max: 0.005
503s  std: 0.001250s
RandomService - process: 0.000s (CPU 0.000s) - finish: 0.000s (CPU 0
.000s)
  wall  mean: 0.000010s  medi: 0.000012s  min: 0.000003s  max: 0.000
016s  std: 0.000005s
  CPU   mean: 0.000011s  medi: 0.000012s  min: 0.000003s  max: 0.000
017s  std: 0.000006s
AModule - process: 0.001s (CPU 0.001s) - finish: 0.000s (CPU 0.000s)
  wall  mean: 0.000324s  medi: 0.000246s  min: 0.000043s  max: 0.000
684s  std: 0.000268s
  CPU   mean: 0.000405s  medi: 0.000277s  min: 0.000056s  max: 0.000
881s  std: 0.000349s
bottleneck - process: 3.003s (CPU 0.010s) - finish: 0.000s (CPU 0.00
0s)
  wall  mean: 1.001107s  medi: 1.001105s  min: 1.001071s  max: 1.001
144s  std: 0.000030s
  CPU   mean: 0.003237s  medi: 0.002458s  min: 0.001846s  max: 0.005
408s  std: 0.001555s
```

The MemoryObserver

Just an example of a simple `Module` (`km3modules.common.MemoryObserver`) to monitor the peak memory usage.

In [32]:
```
class MemoryKiller(kp.Module):
    def configure(self):
        self.buffer = []

    def process(self, blob):
        self.buffer.append(np.random.rand(100000))

        return blob
    def finish(self):
        self.log.critical("All your memory are belong to us!")


pipe = kp.Pipeline()
pipe.attach(km.common.MemoryObserver, every=1000)
pipe.attach(MemoryKiller)
# pipe.drain(30000);
```

Pipeline Configuration using `TOML` files

A configuration file can be used to pass parameters to modules externally.

pipeline.toml

```
[ModuleName]
parameter_1 = value
parameter_2 = another_value

[AnotherModule]
whatever = 23  # a comment
```

In [33]:
```
class AModule(kp.Module):
    def configure(self):
        self.foo = self.get('foo')

    def process(self, blob):
        self.print(self.foo)
        return blob

class BModule(kp.Module):
    def configure(self):
        self.bar = self.get('bar')

    def process(self, blob):
        self.print(self.bar)
        return blob
```

In [34]:
```
!cat python-based-frameworks-for-km3net/pipeline.toml
```

```
[AModule]
foo = 23

[Variation]
foo = "I am a variation of AModule"

[BModule]
bar = [1, 1, 2, 3, 5]
```

In [35]:
```
pipe = kp.Pipeline(configfile="python-based-frameworks-for-km3net/pipe
pipe.attach(AModule)
pipe.attach(BModule)
pipe.attach(AModule, name='Variation')
pipe.drain(2);
```

```
WARNING ++ Pipeline: Keep in mind that the module configuration file
has precedence over keyword arguments in the attach method!

++ Pipeline: Reading module configuration from 'python-based-framewo
rks-for-km3net/pipeline.toml'
Pipeline and module initialisation took 0.002s (CPU 0.002s).
++ AModule: 23
++ BModule: [1, 1, 2, 3, 5]
++ AModule.Variation: I am a variation of AModule
++ AModule: 23
++ BModule: [1, 1, 2, 3, 5]
++ AModule.Variation: I am a variation of AModule
============================================================
2 cycles drained in 0.004369s (CPU 0.004747s). Memory peak: 149.94 M
B
  wall  mean: 0.000979s  medi: 0.000979s  min: 0.000375s  max: 0.001
584s  std: 0.000604s
  CPU   mean: 0.001160s  medi: 0.001160s  min: 0.000435s  max: 0.001
886s  std: 0.000725s
```

Our two main `Dataclasses`: `Table` and `NDArray`

- `kp.Table` : a 2D table (similar to Pandas (https://pandas.pydata.org) `DataFrame`), `np.recarray` conform
- `kp.NDArray` : N-dimensional array, `np.ndarray` conform

Both of them are lightweight wrappers to NumPy arrays, which add meta data for HDF5 I/O

The `Table` (1/3)

In [36]:
```
tab = kp.Table(
    {
        'a': np.random.rand(5),
        'b': True,  # will be expanded
        'c': [1, 1, 2, 3, 5]
    },
    h5loc = "/the_data",
    split_h5 = False,
    name = "The Awesome Data",
)
tab
```

Out[36]:
```
The Awesome Data <class 'km3pipe.dataclasses.Table'> (rows: 5)
```

In [37]:
```
tab.h5loc, tab.split_h5, tab.name
```

Out[37]:
```
('/the_data', False, 'The Awesome Data')
```

The `Table` (2/3)

In [38]:
```
print(tab)
```

```
The Awesome Data <class 'km3pipe.dataclasses.Table'>
HDF5 location: /the_data (no split)
<f8 (dtype: a) = [0.7461731  0.76988027 0.25497894 0.10754764 0.6001
836 ]
|b1 (dtype: b) = [ True  True  True  True  True]
<i8 (dtype: c) = [1 1 2 3 5]
```

In [39]:
```
np.array(tab)
```

Out[39]:
```
array([(0.7461731 ,  True, 1), (0.76988027,  True, 1),
       (0.25497894,  True, 2), (0.10754764,  True, 3),
       (0.6001836 ,  True, 5)],
      dtype=(numpy.record, [('a', '<f8'), ('b', '?'), ('c', '<i8')])
)
```

In [40]:
```
tab.c, tab['c']
```

Out[40]:
```
(array([1, 1, 2, 3, 5]), array([1, 1, 2, 3, 5]))
```

In [41]:
```
tab[2:]
```

Out[41]:
```
The Awesome Data <class 'km3pipe.dataclasses.Table'> (rows: 3)
```

The `NDArray`

Similar to `Table` ...

In [42]:
```
kp.NDArray(np.random.rand(2, 3, 4), h5loc='/the_array', name='The Arra
```

Out[42]:
```
NDArray([[[0.62434947, 0.01142351, 0.87100751, 0.26888005],
          [0.99147295, 0.45715686, 0.02617664, 0.24572004],
          [0.06065855, 0.07437501, 0.5968242 , 0.22847644]],

         [[0.52068617, 0.10966584, 0.8814287 , 0.66015482],
          [0.07412044, 0.47332383, 0.99842233, 0.5996915 ],
          [0.98278568, 0.17175108, 0.53062665, 0.49040183]]])
```

Combining all together: Pipeline + Modules + Blob + Table/NDArray -> HDF5 (1/3)

In [43]:
```
class ThePump(kp.Module):
    def configure(self):
        self.index = 1
    def process(self, blob):
        blob['Foo'] = kp.Table(
            {
                'a': np.random.rand(self.index),
                'b': np.arange(self.index)
            },
            h5loc='/foo'
        )
        self.index += 1
        return blob
```

In [44]:
```
pipe = kp.Pipeline()
pipe.attach(ThePump)
pipe.attach(kp.io.HDF5Sink, filename='output.h5')
pipe.drain(3);
```

```
Pipeline and module initialisation took 0.001s (CPU 0.001s).
++ km3pipe.io.hdf5.HDF5Sink.HDF5Sink: HDF5 file written to: output.h
5
============================================================
3 cycles drained in 0.022964s (CPU 0.021942s). Memory peak: 150.75 M
B
  wall  mean: 0.002499s  medi: 0.001607s  min: 0.001393s  max: 0.004
497s  std: 0.001415s
  CPU   mean: 0.002617s  medi: 0.001608s  min: 0.001523s  max: 0.004
720s  std: 0.001487s
```

Combining all together: Pipeline + Modules + Blob + Table/NDArray -> HDF5 (2/3)

In [45]:
```
!ptdump output.h5
```

```
/ (RootGroup) 'KM3NeT'
/foo (Table(6,), fletcher32, shuffle, zlib(5)) 'Generic Table'
/group_info (Table(3,), fletcher32, shuffle, zlib(5)) 'Group Info'
```

In [46]:
```
import pandas as pd

pd.read_hdf('output.h5', '/foo')
```

Out[46]:

	a	b	group_id
0	0.527019	0	0
1	0.746563	0	1
2	0.844844	1	1
3	0.453001	0	2
4	0.554597	1	2
5	0.685403	2	2

Combining all together: Pipeline + Modules + Blob + Table/NDArray -> HDF5 (3/3)

In [47]:
```
def printer(blob):
    print(blob)
    return blob

pipe = kp.Pipeline()
pipe.attach(kp.io.HDF5Pump, filename='output.h5')
pipe.attach(printer)
pipe.drain();
```

```
++ km3pipe.io.hdf5.HDF5Pump.HDF5Pump: Opening output.h5
Pipeline and module initialisation took 0.005s (CPU 0.006s).
Blob (2 entries):
 'Foo'       => Foo <class 'km3pipe.dataclasses.Table'> (rows: 1)
 'GroupInfo' => GroupInfo <class 'km3pipe.dataclasses.Table'> (rows:
1)
Blob (2 entries):
 'Foo'       => Foo <class 'km3pipe.dataclasses.Table'> (rows: 2)
 'GroupInfo' => GroupInfo <class 'km3pipe.dataclasses.Table'> (rows:
1)
Blob (2 entries):
 'Foo'       => Foo <class 'km3pipe.dataclasses.Table'> (rows: 3)
 'GroupInfo' => GroupInfo <class 'km3pipe.dataclasses.Table'> (rows:
1)
================================================================
3 cycles drained in 0.316570s (CPU 0.854610s). Memory peak: 168.93 M
B
  wall  mean: 0.103696s  medi: 0.000480s  min: 0.000264s  max: 0.310
344s  std: 0.146122s
  CPU   mean: 0.282741s  medi: 0.000554s  min: 0.000334s  max: 0.847
335s  std: 0.399228s
```

Current Development (w.r.t. Pipelines)

- parallel pipelines (using `multiprocessing`)
- additional pipeline stage `prepare()` between `configure()` and first `process()` call

Future Plans for KM3Pipe

- extracting the core pipeline functionality and create a **standalone package**
- flexible **data provenance** for HDF5 output (`HDF5Sink`)
  - include `Pipeline` structure
  - other metadata

Summary

- **aanet**: a convenient way to use (Py)ROOT object in the NumPy world
- **OrcaNet**: deep learning training organiser
- **KM3Pipe**: a multi-purpose framework with a focus on its pipeline feature
  - lightweight pipeline framework with two and a half classes
  - integrated logging
  - performance statistics (CPU time, peak memory)
  - services to expose functionality to other modules
  - configurable pipelines using `TOML` files