



Documentation and user's guide for the packages
PaPy, NuMap, & NuBio/NuBox

Marcin Cieřlik, Cameron Mura
July 2010

PaPy Documentation

Release 1.0

July 30, 2010

CONTENTS

1	Contents	3
1.1	Introduction	3
1.2	Installation	4
1.3	Quick Introduction	9
1.4	Architecture	11
1.5	About the Parallelism in <i>PaPy</i>	21
1.6	Inter Process Communication	29
1.7	The Produce / Spawn / Consume Idiom	30
1.8	Runtime (Logging, Interaction, Errors and Timeouts)	30
1.9	Optimization	31
1.10	Examples	32
1.11	PaPy for Bioinformatics	32
1.12	Workflows	32
1.13	Known Issues	32
1.14	FAQ	33
1.15	Dictionary of terms and definitions	33
1.16	PaPy API	36
2	Indices and tables	47
	Python Module Index	49
	Index	51

A parallel pipeline is a workflow, which consists of a series of connected processing steps to model computational processes and automate their execution in parallel on a single multi-core computer or an ad-hoc grid.

You will find **PaPy** useful if you need to design and deploy a scalable data processing workflow that depends on Python libraries or external tools. **PaPy** makes it reasonably easy to convert existing code bases into proper workflows.

- **project page:** <http://code.google.com/p/papy/>
- **repository lives at:** <http://papy.googlecode.com/svn/trunk/papy>
- **most recent documentation sources:** <http://papy.googlecode.com/svn/trunk/papy/doc/source/>
- **author email:** mpc4p@virginia.edu

This documentation covers the design, implementation and usage of **PaPy**. It consists of a hand-written manual and an API-reference. Please refer also to the rich comments in the source code, examples, workflows and test cases (all included in the source-code distribution).

CONTENTS

1.1 Introduction

Many computational tasks require sequential processing of data i.e. the global data set is split into items which are processed separately by multiple chained processing nodes. This is generally called a dataflow. **PaPy** adapts flow-based programming paradigms and allows to create a data-processing pipeline which allows for both data and task parallelism. In the process of design we tried to make the **PaPy** API as idiosyncrasy-free as possible, relying on familiar concepts of map functions and directed acyclic graphs.

1.1.1 Feature summary

This is a list of features of a workflow constructed using the **PaPy** package and its additional components.

- construction of arbitrarily complex pipelines (any directed acyclic graph is a valid workflow)
- evaluation is lazy-buffered (allows to process datasets that do not fit into memory)
- flexible local and remote parallelism (local and remote resources can be pooled)
- shared local and remote resources (resource pools can be flexibly assigned to processing nodes)
- robustness to exceptions
- support for time-outs
- real-time logging / monitoring
- os-independent (really a feature of `multiprocessing`)
- distributed (really a feature of `RPyC`)
- small code-base
- tested & documented.

1.1.2 Description

Workflows are constructed from components of orthogonal functionality:

- the function wrapping `Workers`
- the connection capable `Pipers`
- the topology defining `Dagger`
- the parallel executors `NuMaps`

The `Dagger` connects `Pipers` via pipes into a directed acyclic graph while the `NuMaps` are assigned to `Pipers` and evaluate their `Workers` either locally using threads or processes or on remote hosts. The `Workers` allow to compose multiple functions while the `Pipers` allow to connect the inputs and outputs of `Workers` as defined by the `Dagger` topology. Data parallelism is possible because data items are independent i.e. if it is a collection (or can be split into one) of data-items: files, messages, sequences, arrays. **PaPy** enables task parallelism by allowing the data processing functions to be evaluated on different computational resources represented by `NuMap` instances.

PaPy is written in and for Python this means that the user is expected to write Python functions with defined call/return signatures, but the function code is largely arbitrary e.g. they can call a perl script or import a library. **PaPy** focuses on modularity, functions should be re-used and composed within pipelines and `Workers`.

The **PaPy** workflow automatically logs its execution is resistant to exceptions and timeouts and should work on all platforms where `multiprocessing` is available. It also allows to compute over a cross-platform ad-hoc grid using the **RPyC** package.

1.1.3 Where/When should PaPy be used?

It is likely that you will benefit from using **PaPy** if some of the following is true:

- you need to process large collections of data items.
- your data collection is too large to fit into memory.
- you want to utilize an ad-hoc grid.
- you have to construct a complex workflow or data-flows.
- you are likely to deal with timeouts or bogus data.
- the execution of your workflow needs to be logged / monitored.
- you want to refactor existing code.
- you want to reuse (wrap) existing code.

1.1.4 Where/When should PaPy not be used?

- You do not need a workflow just a method to evaluate a function in parallel (consider `NuMap` for this).
- The parallel evaluation will improve performance only if the functions have sufficient granularity i.e. a computation to communication ratio.
- Your input is not a collection and it does not allow for data parallelism.

1.2 Installation

This guide will go through the steps required to install a bleeding-edge version of **PaPy** on a UNIX machine using the `bash` shell. The user is left with two options to install **PaPy** globally or into a sandbox and to use a stable version or possibly unstable sources.

An `easy_install` assumes that your default interpreter is Python 2.6, you have `setuptools` installed and you want to install **PaPy** into system-wide site-packages (the location where Python looks for installed libraries). If the default Python interpreter for your operating system is different from Python 2.6 or you do not want to put **PaPy** or its dependencies in the system directories or finally you'd like the latest source-code revision of **PaPy** read further choose the fancy way of installing **PaPy**.

1.2.1 Installing PaPy the easy way

PaPy is indexed on the **PyPI** (Python Package Index). And can be installed simply via:

```
$ su -c "easy_install papy"
```

This will install the `papy`, `numap`, `nubio`, `nubox` and `rpyc` packages. If this did not work please try to use the manual source code distributions.

- download **PaPy**, **NuMap**, **NuBio**, **NuBox**, **RPyC** snapshots from <http://muralab.org/papy>
- install it via `easy_install`:

```
$ su -c "easy_install rpyc_YYY.tar.gz"
$ su -c "easy_install numap_YYY.tar.gz"
$ su -c "easy_install nubox_YY.tar.gz"
$ su -c "easy_install nubio_YYY.tar.gz"
$ su -c "easy_install papy_YYY.tar.gz"
```

replacing XXX with the version numbers of the downloaded files.

Optional dependencies to install/build/deploy **PaPy** are:

- `easy_install` (`setuptools`)
- **Sphinx** (`sphinx`)

PaPy and Python development is much easier using the following tools:

- `virtualenv`
- `virtualevnwrapper`

1.2.2 Installing PaPy the fancy way

The fancy (and cleaner) way of using **PaPy** is to create a virtual environment to use **PaPy**. The general stream of action is to install Python 2.6 (if required) install `setuptools` for Python 2.6 (if required). Later we create sandbox environment by using `virtualenv` and `virtualenvwrapper` that is finally populated with **PaPy** and dependencies checked-out from a source repository. This guide assumes you are using `bash` and a recent version of something UNIX-like.

Getting Python

Most Linux distributions and Mac OSX ship a recent version of Python. You can (and should) skip this step if you have Python 2.6 and it is the default Python interpreter. To check this open a shell and type:

```
$ python
```

This should return something similar to this:

```
Python 2.6.2 (r262:71600, Jun 12 2009, 10:38:05)
[GCC 4.1.2 (Gentoo 4.1.2 p1.1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

Notice the version in the first line. If it is not 2.6.x or higher, but not 3.X skip this section as you already have a supported Python version. Otherwise it might be that your operating system still provides you a Python 2.6, but the name of the executable is different try:

```
$ python2.6
```

If, and only if, this fails with `python2.6: command not found` or something similar proceed with the manual system-wide installation of Python 2.6 below.

This installation will not change the default Python interpreter for your distribution. A compiled version of Python 2.6 for your distribution might be available from official distribution or third-party repositories, please check them before instead of proceeding with the following os-independent method.

1. Download Python 2.6 from <http://www.python.org/download/> you want a source tarball and the highest 2.6.x version available. You do not want to install Python 3.0.x or 3.1.x
2. Unpack the compressed tarball into any directory e.g.:

```
tar xfv Python-2.6.2.tgz
```

3. go to the root of the unpacked directory and compile Python2.6:

```
cd ./configure --prefix=/usr make su -c "make altinstall"
```

this will install an executable `python2.6` into `/usr/bin`

4. try if it worked:

```
$ python2.6
```

Be careful! Some of the following steps are different depending on whether you built your own Python 2.6 distribution or your system is using Python 2.6 by default. Make sure you always type `python2.6` not `python` to be on the safe side.

Getting `setuptools`

We will use `setuptools` to install **PaPy**, the tools to necessary create a sandbox environment and **PaPy**'s dependencies. This is a very common package and is most likely already installed. But we need it for explicitly for Python 2.6. To test if it is there it try:

```
$ easy_install2.6
```

If you encounter an error, and you did **not** build Python 2.6 manually you should try to install it from your distribution's repository. On Gentoo the package is called "dev-python/setuptools" on Fedora it is "python-setuptools" and "python-setuptools-devel" for Ubuntu the package is called "python-setuptools" You can install them by e.g.:

```
# Gentoo
$ su -c "emerge setuptools"
# Ubuntu
$ sudo apt-get install python-setuptools
# Fedora
$ su -c "yum install python-setuptools"
```

If you had to manually compile Python 2.6 the standard distribution `setuptools` package are most likely installed but only for the default system-wide Python e.g. Python 2.5. You will have to install `setuptools` manually for the just built and installed Python 2.6 interpreter.

1. Download `setuptools` from <http://pypi.python.org/pypi/setuptools> you will want the latest source version at the time of writing it is `setuptools-0.6c11.tar.gz`.
2. Unpack the compressed tarball into any directory:

```
$ tar xvf setuptools-0.6c11.tar.gz
```

3. Go to the root of the extracted directory:

```
$ cd setuptools-0.6c11
```

4. Now we install setuptools using the python 2.6 executable, but first we have to make sure that we don't override /usr/bin/easy_install. If setuptools is by default installed for a different Python interpreter. If there is no other Python interpreter or you do not care you can skip the following and just issue:

```
# python2.6 setup.py install
```

To prevent overriding /usr/bin/easy_install we edit the setup.py file:

```
<snip>
"console_scripts": [
    "easy_install = setuptools.command.easy_install:main",
    "easy_install-%s = setuptools.command.easy_install:main"
    % sys.version[:3]],
<snip>
```

by commenting out the second line i.e.:

```
<snip>
"console_scripts": [
#    "easy_install = setuptools.command.easy_install:main",
    "easy_install-%s = setuptools.command.easy_install:main"
    % sys.version[:3]],
<snip>
```

5. now we can safely run the installation:

```
$ python2.6 setup.py install
```

6. and verify that we have to type easy_install-2.6:

```
# CORRECT
$ easy_install-2.6
error: No urls, filenames, or requirements specified (see --help)
# NOT CORRECT
-bash: easy_install-2.6: command not found
```

Creating a virtual environment

Generally we do not want to pollute the system-wide distribution with **PaPy** and its dependencies, but we can and this step is optional, although maintenance of **PaPy** might be easier in a virtual environment. We will create a virtual environment just for **PaPy**. We will install virtualenv and virtualenvwrapper into the newly created Python installation or standard Python2.6 using easy_install-2.6.:

```
$ su -c "easy_install-2.6 virtualenv"
$ su -c "easy_install-2.6 virtualenvwrapper"
```

Note that these packages are installed system-wide. Now we have to configure virtualenvwrapper on a per-user basis. We have to edit the .bashrc file.

1. determine where the wrapper got installed:

```
$ which virtualenvwrapper.sh
```

2. create a directory where you will hold the virtual environment(s):

```
$ mkdir $HOME/.virtualenvs
```

3. add the following two lines to `~/.bashrc` replace `__REPLACE_ME__` with whatever the output from the first command was.:

```
export WORKON_HOME=$HOME/.virtualenvs
source __REPLACE_ME__
```

Now we have to source the edited `.bashrc` file:

```
$ source ~/.bashrc
```

This should not generate any errors. We are finally ready to create a virtual Python2.6 environment for **PaPy**:

```
$ mkvirtualenv -p python2.6 --no-site-packages papy26
```

This will install a clean virtual environment called `papy26` and activate it. Working with virtual environments is easy. To use it type `workon papy26` to leave it type `deactivate`.

Installing PaPy dependencies and tools

First we have to switch to the virtual environment to host all **PaPy** related code:

```
$ workon papy26
```

Next we install various libraries on which **PaPy** depends. The general install command is:

```
$ easy_install-2.6 PACKAGE_NAME
```

You do not have to be root to install the packages into the virtual environment:

1. installing RPyC to use **PaPy** on a grid:

```
$ easy_install-2.6 papy
```

2. install Sphinx to build **PaPy** documentation:

```
$ easy_install-2.6 sphinx
```

If the above did not work because e.g. some of the tarfiles could not be downloaded we have to download snapshots manually from <http://muralab.org/papy/downloads> to install the libraries. The general installation command is:

```
$ easy_install-2.6 PACKAGE_NAME.tar.gz
```

The required packages are `rpyc`, `nubox`, `numap` and `nubio`.

1.2.3 Develop PaPy

To install **PaPy** from sources please follow the instructions for installing **PaPy** the fancy way and create a `papy26_dev` sandbox and install `rpyc` only (we will use `papy`, `nubox`, `nubio` and `numap` from the repository).

1. make sure you have subversion:

```
$ svn
Type 'svn help' for usage.
```

If this returns an error you have to install the subversion package:

```
# on Gentoo
$ su -c "emerge subversion"
# on Fedora
$ su -c "yum install subversion"
# On Ubuntu
$ sudo apt-get install subversion
```

2. create a workspace to hold the source code

```
$ mkdir ~/workspace
```

3. check-out all the necessary sources:

```
$ http://muralab.org/papy/workspace
```

4. switch to the papy26_dev environment:

```
$ workon papy26_dev
```

5. add all the required source code directories:

```
$ add2virtualenv ~/workspace/papy/src
$ add2virtualenv ~/workspace/numap/src
$ add2virtualenv ~/workspace/nubio/src
$ add2virtualenv ~/workspace/nubox/src
```

6. Verify it worked.:

```
$ python2.6
>>> import papy
>>> import nubio
>>> import nubox
>>> import NuMap
```

1.3 Quick Introduction

Warning: Parallel features of **PaPy** do **not** work in the interactive interpreter. This is a limitation of the Python multiprocessing module.

This part of the documentation is all about creating a valid **PaPy** workflow.

Creating a workflow involves several steps:

1. Writing worker functions (the functions in the nodes)
2. Creating `Worker` instances (to specify worker function parameters)
3. (optionally) creating `NuMap` instances to represent parallel computational resources.
4. Creating `Piper` instances (to specify how/where to evaluate the workers)
5. Creating a `Dagger` instance (to specify the workflow connections)

After construction a workflow can be deployed (or run) by connecting it to input data and retrieving batches of results.

Workflows exist in two very distinct states before and after they are started. Those states correspond to the “construction-time” and “run-time” of a workflow lifecycle. At “creation-time” functions are defined (written or imported) and the data-flow is defined by connecting functions by directed pipes. At run time data is actually “pumped” through the pipeline. A pipeline can be saved and loaded as a python script.

1.3.1 The Workflow Graph

In PaPy a workflow is a directed acyclic graph. The direction of the graph is defined by the data flow (pipes) between the data-processing units (nodes, `Pipers`). PaPy pipelines can be branched i.e. two downstream `Pipers` might consume input from the same upstream `Piper` or one down-stream `Piper` consumes data from several upstream `Pipers`. `Pipers`, which consume (are connected to) data from outside the directed acyclic graph are input `Pipers`, while `Pipers` to which no other `Pipers` are connected to are output `Pipers`. PaPy supports pipelines with multiple inputs and outputs, also several input nodes can consume the same external data.

1.3.2 Workflow Execution

Workflows process data items either in parallel or lazily one after another. Distributed evaluation is enabled by the `numap` package that provides the `NuMap` object. Instances of this object represent computational resources (pools of local and/or remote processes) that are assigned to processing nodes. It is possible that multiple processing nodes share a single resource pool.

1.3.3 Creating a workflow

Our first workflow

Writing a worker function

PaPy imposes restrictions on the inputs and outputs of a worker function. Further we recommend a generic way to construct a pipeline:

```
input_iterator -> input_piper -> ... processing pipers ... -> output_piper
```

The `input_piper` should transform input data items into a specific Python objects. Processing pipers should manipulate or transform this objects finally the `output_piper` should be used to store the output of the workflow persistently i.e. it should return `None`. PaPy provides useful functions to create input/output pipers.

We will start by writing the simplest possible function it will do nothing, but return the input:

```
def pass_input(inbox):
    input = inbox[0]
    return input
```

This function can only be used in a `Piper` that has only one input i.e. it uses only the first element of the “inbox” tuple. A function with two input values could be used at a point in a workflow where two branches join.:

```
def merge_inputs(inbox):
    input1 = inbox[0]
    input2 = inbox[1]
    return (input1, input2)
```

Writing a worker function which takes arguments

A function can be written to take additional arguments (with or without default values). In the first example the first element in the `inbox` is multiplied by a constant factor specified as the `number` argument. In the second example an arithmetical operation will be done on two numbers as specified by the `operation` argument.:

```
def multiply_by(inbox, number):
    input = inbox[0]
    output = input * number
    return output

def calculator(inbox, operation):
    input1 = inbox[0]
    input2 = inbox[1]
    if operation == 'add':
        result = input1 + input2
    elif operation == 'subtract':
        result = input1 - input2
    elif operation == 'multiply':
        result = input1 * input2
    elif operation == 'divide':
        result = input1 / input2
    else:
        result = 'error'
    return result
```

The additional arguments for these functions are specified when a `Worker` is constructed:

```
multiply_by_2 = Worker(multiply_by, number =2)
calculate_sum = Worker(calculator, operation ="sum")
```

The argument names need not to be given:

```
multiply_by_3 = Worker(multiply_by, 3)
calculate_product = Worker(calculator, "multiply")
```

If a worker is constructed from multiple functions i.e. “sum the two inputs and multiply the result by 2”:

```
# positional arguments
sum_and_multiply_by_2 = Worker((calculator, multiply_by), \
                               (('sum',), (3,))):

# keyworded arguments
sum_and_multiply_by_2 = Worker((calculator, multiply_by), \
                               kwargs = \
                               ({'operation':'sum'}, {'number':3}))
```

In the last example the second argument given is in the first version a tuple of tuples which are the positional arguments for the function or as in the second example a tuple of dictionaries with named arguments.

1.4 Architecture

The architecture of **PaPy** is remarkably simple and intuitive yet flexible. It consists of only four core components (classes) to construct a data processing pipeline. Each component provides an isolated subset of the functionality, which includes defining the: processing nodes, connectivity and computational resources of a workflow and further enables deployment and run-time interactions (e.g. monitoring).

PaPy is very modular, functions can be used in several places in a pipeline or re-used in another pipelines. Computational resources can be shared among workflows and processing nodes.

In this chapter we first introduce object-oriented programming in the context of **PaPy**, explain briefly the core components (building blocks). In later sections we revisit each component and explain the how and why.

1.4.1 Understanding the object-oriented model

PaPy is written in an object-oriented(OO) way. The main components: Plumber, Dagger, Pipers and Workers are in fact class objects. For the end-user it is important to distinguish between classes and class instances. In Python both classes and class instances are objects. When you import the module in your script:

```
import papy
```

A new object (a module) will be available i.e. you will be able to access classes and functions provided by **PaPy** e.g.:

```
papy.SomeClass
```

The name of the imported object will be `papy`. This object has several attributes which correspond to the components and interface of `papy` e.g.:

```
papy.Plumber
papy.Dagger
papy.Piper
papy.Worker
```

Attributes are accessed in Python using the `object.attribute` notation. These components are classes not class instances. They are used to construct class instances which correspond to the run-time of the program. A single class can in general have multiple instances. A class instance is constructed by “calling” (in fact initializing) the class i.e.:

```
class_instance = Class(parameters)
```

The important part is that using `papy` involves constructing class instances.:

```
worker_instance = Worker(custom_function(s), argument(s))
piper_instance = Piper(worker_instance, options)
your_interface = Plumber(options)
```

1.4.2 core components

The core components form the end-user interface i.e. the classes which the user is expected use directly.

- **NuMap - An implementation of an iterated map function which can process** multiple tasks (function-sequence tuples) in parallel using either threads or processes on the local machine or on remote **RPyC** servers. NuMap instances represent computational resources.
- **Pipers(Workers) - combined define the processing nodes by wrapping** user-defined functions and handling exceptions.
- **Dagger - defines the connectivity of the pipeline in the form of a directed** acyclic graph i.e. the connectivity of the flow (pipes).
- **Plumber - provides the interface to set-up run and monitor a workflow at** run-time.

1.4.3 The NuMap class

The NuMap class is provided by the separate module `numap` and is described further in the section about parallel and distributed workflows. Here it suffices to say that it is an object which models a pool of computational resources and allows to execute **multiple** functions using a shared pool of local or remote of workers. A NuMap can be used in any python code as an alternative to `multiprocessing.Pool` or `itertools.imap`. For details please refer to the documentation and API for `numap`.

object provides a method to evaluate a functions on a sequence of changing arguments provided with optional positional and keyworded arguments to modify the behaviour of the function. Just

like `multiprocessing.Pool.imap` or `itertools.imap` with the key differences that unlike `itertools.NuMap` it evaluates results in parallel. Compared to `multiprocessing.Pool.imap` it supports multiple functions (called tasks), which are evaluated not one after another, but in an alternating fashion. `NuMap` is completely independent from PaPy and can be used separately (it is a standalone package).

In PaPy the lazy `imap` functions is replaced with a pool implementation `NuMap`, which allows for a parallelism vs. memory requirements trade-off.

1.4.4 The Worker class

The `Worker` is a class which is created with a function or multiple functions (and the functions arguments) as arguments to the constructor. It is therefore a function wrapper. If multiple functions are supplied they are assumed to be nested with the last function being the outer most i.e.:

```
(f, g, h) is h(g(f()))
```

If a `Worker` instance is called this composite function is evaluated on the supplied argument.:

```
from papy import Worker
from math import radians, degrees
def papy_radians(input):
    return radians(input[0])
def papy_degrees(input):
    return degrees(input[0])
worker_instance = Worker((papy_radians, papy_degrees))
worker_instance([90.])
90.0
```

In this example we have created a composite `Worker` from two functions `papy_radians` and `papy_degrees`. The first function converts degrees to radians the second converts radians to degrees. Obviously if those two functions are nested their result is identical to their input. `papy_radians` is evaluated first and `papy_degrees` second so the result is in degrees.

The `Worker` performs several functions:

- standardizes the inputs and outputs of nodes.
- allows to reuse and combine multiple functions into as single node
- catches and wraps exceptions raised within functions.
- allows functions to be evaluated on remote hosts.

A `Worker` expects that the wrapped function has a defined input and output signature. The input is expected to be boxed in a tuple relative to the output, which should not be boxed. For example the `Worker` instance expects `[item]`, but returns just `item`. Any function which conforms to this is a valid `Worker` function. Most built-in functions need to be wrapped. Please refer to the API documentation and examples on how to write `Worker` functions.

If an exception is raised within any of the user written functions it is caught by the `Worker`, but is **not** raised, instead it is wrapped as a `WorkerError` exception and returned.

The functionality of a `Worker` instance is defined by the functions it is composed of and their arguments. Two `Workers` which are composed of the same functions **and** are called with the same arguments are functionally identical and a single `Worker` instance could replace them i.e. be used in multiple places of a pipeline or in other words in multiple `Piper` instances.

The functions within a `Worker` instance might not be evaluated by the same process as the process that created (and calls) the `Worker` instance. This is accomplished by the `RPyC` package and `multiprocessing` module. A `Worker` knows how to inject its functions into a `RPyC` connection instance, after this the worker method will be called in the local process, but the wrapped functions on the remote host.

```
import rpyc # import the RPyC module from papy
import Worker
power = Worker(pow, (2,)) # power of two
power([2]) # evaluated locally
4
conn = rpyc.classic.connect("some_host")
power._incject(conn) # replace pow with remot pow
power([3]) # evaluated remotely
9
```

A function can run on the remote host i.e. remote Python process/thread only if the modules on which this function depends are available on that host and they are imported. NuMap provides means to attach import statements to function definitions using the `imports` decorator. In this way code sent to the remote host will work if the imported module is available remotely.:

```
@imports(['re'])
def match_string(input, string):
    unboxed = input[0]
    return re.match(string, unboxed)
```

The above example shows a valid worker function with the equivalent of the import statement attached.:

```
import re
```

The `re` module will be available remotely in the namespace of this function i.e. other injected functions might not have access to `re`. For more information see the NuMap documentation.

1.4.5 Built-in worker functions

Several classes of `Worker` functions are already part of **PaPy**. This collection is expected to grow, currently the following types of workers are included.

- `core` - basic data-flow
- `io` - serialisation, printing and file operations

These are available in the `papy.util.func` module. This includes the family of `passer` functions. They do not alter the incoming data, but are used to pass only streams from certain input pipes. For example a `Piper` connected to 3 other `Pipers` might propagate input from only one.

- `ipasser` - propagates the “i”th input pipe
- `npasser` - propagates the “n”-first input pipes
- `spasser` - propagates the pipes with numbers in “s”

For example:

```
from papy.util.func import *
worker = Worker(ipasser, (0,)) # passes only the first pipe
worker = Worker(ipasser, (1,)) # passes only the second pipe
worker = Worker(npasser, (2,)) # passes the first two pipes
worker = Worker(spasser, ((0,1),)) # passes pipes 0 and 1
worker = Worker(spasser, ((1,0),)) # passes pipes 1 and 0
```

The output of the passes is a *single* tuple of the passed pipes:

```
input0 = [0,1,2,3,4,5]
input1 = [6,7,8,9,10,11]

worker = Worker(spasser, (1,0))
# will produce output
[(6,0), (7,1), ...]
```

Functions dealing with input/output relations i.e. data storage and serialization currently allow serialization using the pickle and JSON protocols and file-based data storage.

Data serialization is a way to convert objects (and in Python almost everything is an object) into a sequence, which can be stored or transmitted. **PaPy** uses the `pickle` serialization format to transmit data between local processes and `brine` (an internal serialization protocol from `RPyC`) to transmit data between hosts. The user might however want to save and load data in a different format.

1.4.6 Writing functions for Workers

A worker is an instance of the class `Worker`. `Worker` instances are created by calling the `Worker` class with a function or several functions as the argument. optionally an argument set (for the function) or argument sets (for multiple functions) can be supplied i.e.:

```
worker_instance = Worker(function, argument)
```

or:

```
worker_instance = Worker(list_of_functions, list_of_arguments)
```

A worker instance is therefore defined by two elements: the function or list of functions and the argument or list of arguments. This means that two different instances which have been initialized using the same functions *and* respective arguments are functionally equal. You should think of worker instances as nested curried functions (search for “partial application”).

Writing functions suitable for workers is very easy and adapting existing functions should be the same. The idea is that any function is valid if it conforms to a defined input/output scheme. There are only few rules which need to be followed:

1. The first input argument: each function in a worker will be given a n-tuple of objects, where n is the number input iterators to the `Worker`. For example a function which sums two numbers should expect a tuple of length 2.
2. Remember python uses 0-based counting. If the `Worker` has only one input stream the input to the function will still be a tuple i.e. a 1-tuple.
2. The additional (and optional) input arguments: a function can be given additional arguments.
3. The output: a function should return a single object *_not_* enclosed in a wrapping 1-tuple. If a python function has no explicit return value it implicitly returns `None`.

Examples:

single input, single output:

```
def water_to_water(inp):
    result = inp[0]
    return result
```

single input, no explicit output:

```
def water_to_null(inp):
    null = inp[0]
```

multiple input, single output:

```
def water_and_wine(inp):
    juice = inp[0] + inp[1]
    return juice
```

multiple input, single output, parameters:

```
def water_and_wine_dilute(inp, dilute =1):
    juice = inp[0] * dilute + inp[1]
    return juice
```

Note that in the last examples `inp` is a 2-tuple i.e. the Piper based on such a worker/function will expect two input streams or in other words will have two incoming pipes. If on the other hand we would like to combine elements in the input/object from a single pipe we have to define a function like the following:

```
def sum2elements(inp):
    unwrapped_inp = inp[0]
    result = unwrapped_inp[0] + unwrapped_inp[1]
    return result
```

In other words the function receives a wrapped object but returns an unwrapped. All python objects can be used as results except Exceptions. This is because Exceptions are not evaluated down-stream but are passively propagated.

1.4.7 Writing functions for output workers

An output worker is a worker, which is used in a piper instance at the end of a pipeline i.e. in the last piper. Any valid worker function is also a valid output worker function, but it is recommended for the last piper to persistently save the output of the pipeline. The output worker function should therefore store its input in a file, database or eventually print it on screen. The function should not return data. The reason for this recommendation are related to the implementation details of the IMap and Plumber objects.

1. The Plumber instance runs a pipeline by retrieving results from output pipers *without* saving or returning those results
2. The IMap instance will retrieve results from the output pipers *without* saving whenever it is told to stop *before* it consumed all input.

The latter point requires some explanation. When the stop method of a running IMap instance is called the IMap does not stop immediately, but is scheduled to stop after the current stride is finished for all tasks. To do this the output of the pipeline has to be 'cleared' which means that results from output pipers are retrieved, but not stored. Therefore the 'storage' should be a built-in function of the last piper. An output worker function might therefore require an argument which is a connection to some persistent storage e.g. a file-handle.

1.4.8 The Piper class

A Piper instance represents a node in the directed graph of the workflow. It defines what function(s) should at this node be evaluated (via the supplied Worker instance) and how they should be evaluated (via the optional NuMap instance, which defines the uses computational resources). Besides that it performs additional functions which include:

- logging and reporting
- exception handling
- timeouts
- produce/spawn/consume schemes

To use a Piper outside a workflow three steps are required:

- creation - requires a Worker instance, optional arguments e.g. a NuMap instance. (`__init__` method)
- connection - connects the Piper to the input. (`connect` method)
- start - allows the Piper to return results, starts the evaluation in NuMap. (`start` method)

In the first step we define the Worker which will be evaluated by the Piper and the NuMap resource to do this computation. Computational resources are represented by NuMap instances. An NuMap instance can utilize local or remote threads or processes. If no NuMap instance is given to the constructor the `itertools.imap` function will be used instead. This function will be called by the Python process used to construct and start the PaPy pipeline.

PaPy has been designed to monitor the execution of a workflow by logging at multiple levels and with a level of detail which can be specified. It uses the built-in Python logging (the `logging` module). The `NuMap` function, which should at this stage be bug free logs only `DEBUG` statements. Exceptions within `Worker` functions are wrapped as `WorkerError` exceptions, these errors are logged by the `Piper` instance, which wraps this `Worker` (a single `Worker` instance can be used by multiple `Pipers`). By default the pipeline is robust to `WorkerErrors` and these exceptions are logged, but they do not stop the flow. In this mode if the called `Worker` instance returns a `WorkerError` the calling `Piper` instance wraps this error as a `PiperError` and **returns** (not raises) it downstream into the pipeline. On the other end if a `Worker` receives a `PiperError` as input it just propagates it further downstream i.e. it does not try meaningless calculations on exceptions. In this way errors in the pipeline propagate downstream as place holder `PiperErrors`.

A `Piper` instance evaluates the `Worker` either by the supplied `NuMap` instance (described elsewhere) or by the builtin `itertools.imap` function (default). In reality after a `Piper` is connected to the input it creates a task i.e. function, data, arguments tuples, which are added to the `NuMap` instance used to call the `imap` function.

`NuMap` instances support timeouts via the optional `timeout` argument supplied to the next method. If the `NuMap` is not able to return a result within the specified time it raises a `TimeoutError`. This exception is caught by the `Piper` instance which expects the result, wrapped into a `PiperError` exception and propagated down-stream exactly like `WorkerErrors`. If the `Piper` is used within a pipeline and a `timeout` argument given the `skipping` argument should be set to `true` otherwise the number of results from a `Piper` will be bigger then the number of tasklets, which will hang the pipeline.:

```
# valid with or without timeouts
universal_piper = Piper(worker_instance, parallel =imap_instance, skipping =True)
# valid only with timeouts
nontimeout_piper = Piper(worker_instance, parallel =imap_instance, skipping =False)
```

Note that the timeouts specified here are ‘computation time’ timeouts. If for example a worker function waits for a server response and the server response does not arrive within some timeout (which can be an argument for the `Worker`) then if this exception is raise within the function it will be wrapped into a `WorkerError` and returned not raised as `TimeoutErrors`.

A single `Piper` instance can only be used once within a pipeline (this is unlike `Worker` instances). `Pipers` are created first and connected to the input data later. The latter is accomplished by their `connect` method.:

```
piper_instance.connect(input_data)
```

If the `Piper` is used within a **PaPy** pipeline i.e. a `Dagger` or `Plumber` instance the user does not have to care about connecting individual `Pipers`. A `Piper` can only be started or disconnected if it has been connected before.:

```
piper_instance.connect(input_data)
piper_instance.disconnect()
# or
piper_instance.start()
```

After starting a `Piper` the tasks are submitted to the thread/process workers in the `NuMap` instance and they are evaluated. This is a process that continues until either the memory “buffer” is filled or the input is consumed. Therefore a `Piper` cannot be simply disconnected when it is “running”. A special method is needed to tell the `NuMap` instance to stop input consumption. Because `NuMap` instances are shared among `Pipers` such a stop can only occur at “stride” boundaries, which are batches of data traversing the workflow. The `Piper` stop method will eventually stop the `NuMap` instance and put the `Piper` in a stopped state that allows the `Piper` to be disconnected.:

```
piper_instance.start()
piper_instance.stop()
piper_instance.disconnect() # can be connected and started
```

Because the stop happens at “stride” boundary data is not lost during a stop. This can be illustrated as follows:

```
#           plus2           plus1
# [1,2,3,4] -----> [3,4,5,6] -----> [4,5,6,7]
# which is equivalent to the following:
# plus1(plus2([1,2,3,4]))
```

If the Pipers `plus2` and `plus1` share a single `Numap` and the “stride” is 2 then the order of evaluation can be (if the results are retrieved):

```
temp1 = plus2(1)
temp2 = plus2(2)
plus1(temp1)
plus1(temp2)
<<return>>
<<return>>
temp1 = plus2(3)
temp2 = plus2(4)
plus1(temp1)
plus1(temp2)
<<return>>
<<return>>
```

Now let’s assume the the stop method has been called just after `plus2(1)`. We do not want to loose the `temp1` result (as 1 has been already consumed from the input iterator and iterators cannot rewind), but we can achieve this only if `plus1(temp1)` is evaluated this in turn (due to the order of evaluation) can happen only after `plus2(2)` has been evaluated (i.e. 2 consumed from the input iterator). To not loose `temp2` `plus1(temp2)` has to be evaluated and finally the evaluation can stop.:

```
temp1 = plus2(1)
temp2 = plus2(2)
plus1(temp1)
plus1(temp2)
(stopped)
```

After the stop method returns all worker processes/threads and helper threads return (join) and the user can close the Python interpreter.

It is **very** important to realize what happens with the two calculated results. As has been already mentioned a proper **PaPy** pipeline should have an output `Piper` i.e. a one that persistently stores the result.

1.4.9 The Dagger

The `Dagger` is an object to connect `Piper` instances into a directed acyclic graph (DAG). It inherits most methods of the `DictGraph` object, which is a concise implementation of a graph data-structure. The `DictGraph` instance is a dictionary of arbitrary hashable objects i.e. the “object nodes” e.g. a `Piper`. The values for the objects are instances of the `Node` class i.e. “topological nodes”. A “topological node” instance is a also dictionary of “object nodes” and their corresponding “topological nodes”. An “object node”(A) of the `DictGraph` is contained in a “topological node” corresponding to another “object node”(B) if there exist an edge from (A) to (B). A and B might even be the same “object node” (self-loop). A “topological node” is therefore a sub-graph of the `DictGraph` instance centered around a “object node” and the whole `DictGraph` is a recursively nested dictionary. The `Dagger` is designed to store `Piper` instances as “object nodes” and provides additional methods, whereas the `DictGraph` makes no assumptions about the `object` type.

Edges vs. pipes

A `Piper` instance is created by specifying a `Worker` (and optionally `Numap` instance) and needs to be connected to an input. The input might be another `Piper` or any Python iterator. The output of a `Piper` (upstream) can be

consumed by several `Pipers` (downstream), while a `Piper` (downstream) might consume the results of multiple `Pipers` (upstream). This allows `Pipers` to be used as arbitrary nodes in a directed acyclic graph the `Dagger`.

To be precise the direction of the edges is opposite to the direction of the data stream (pipes). Upstream `Pipers` have incoming edges from downstream `Pipers` this is represented as a pipe with a opposite orientation i.e. upstream \rightarrow downstream.

As a result of the above it is much more natural to think of connections between `Pipers` in terms of data-flow upstream \rightarrow downstream (data flows from upstream to downstream) then dependency downstream \rightarrow upstream (downstream depends on upstream). The `DictGraph` represents dependency information as directed edges (downstream \rightarrow upstream), while the `Dagger` class introduces the concept of pipes to ease the understanding of **PaPy** and make mistakes less common. A pipe is nothing else then a reversed edge. To make this explicit:

```
input  $\rightarrow$  piper0  $\rightarrow$  piper1  $\rightarrow$  output #  $\rightarrow$  represents a pipe (data-flow)
input  $\leftarrow$  piper0  $\leftarrow$  piper1  $\leftarrow$  output #  $\leftarrow$  represents an edge (dependency)
```

The data is stored internally as edges, but the interface uses pipes. Method names are explicit.:

```
dagger_instance.add_edge() # inherited expects and edge as input
dagger_instance.add_pipe() # expects a pipe as input
```

Note: Although all `DictGraph` methods are available from the `Dagger` the end-user should use `Dagger` specific methods. For example the `DictGraph` method `add_edge` will allow to add any edge to the instance, whereas `add_pipe` method will not allow to introduce cycles.

Working with the Dagger

Creation of the a `Dagger` instance is very easy. An empty `Dagger` instance is created without any arguments to the constructor.:

```
dagger_instance = Dagger()
```

Optionally a set of `Pipers` and/or pipes can be given:

```
dagger_instance = Dagger(sequence_of_pipers, sequence_of_pipes)
# which is equivalent to:
dagger_instance.add_pipers(sequence_of_pipers)
dagger_instance.add_pipes(sequence_of_pipes)
# a sequence of pipers allows to easily add branches
dagger_instance.add_pipers([1, 2a, 3a, 4])
dagger_instance.add_pipers([1, 2b, 3b, 4])
# in this example a Dagger will have 6 pipers (1, 2a, 2b, 3a, 3b, 4), one
# branch point 1, one merge point 4, and two branches (2a, 3a) and (2b, 3b).
```

The `Dagger` allows to add/delete `Pipers` and pipes:

```
dagger_instance.add_piper(`Piper`)
dagger_instance.del_piper(`Piper` or piper_id)
dagger_instance.add_pipers(pipers)
dagger_instance.del_pipers(pipers or piper_ids)
```

The id of a `Piper` is a run-time specific number associated with a given `Piper` instance. This number can be obtained by calling the built-in function `id`:

```
id(`Piper`)
```

This number is also shown when a `Piper` instance is printed.:

```
print piper_instance
```

or represented:

```
repr(piper_instance)
```

The representation of a `Dagger` instance also shows the id of the `Pipers` which are contained in the workflow.:

```
print dagger_instance
```

The id of a `Piper` instance is define at run-time (it corresponds to the memory address of the object) therefore it should not be used in scripts or saved in any way. Note that the lenght of this number is platform-specific and that no guarantee is made that two `Pipers` with non-overlapping will not have the same id. The resolve method:

```
dagger_instance.resolve(`Piper` or piper_id)
```

returns a `Piper` instance if the supplied `Piper` or a `Piper` with the supplied id is contained in the `dagger_instance`. This method by default raises a `DaggerError` if the `Piper` is not found. If the argument `forgive` is `True` the method returns `None` instead:

```
dagger_instance.resolve(missing_piper) # raise DaggerError
dagger_instance.resolve(missing_piper, forgive =True) # returns None
```

The Dagger run-time

The run-time of a `Dagger` instance begins when it's start method is called. A `Dagger` can only be started if it is connected. Connecting a `Dagger` means to connect all `Pipers` which it contains as defined by the pipes in the `Dagger`. After the `Dagger` is connected it can be started, starting a `Dagger` means to start all it's `Pipers`. `Pipers` have to be started in the order of the data-flow i.e. a `Piper` can only be started after all it's up-stream `Pipers` have been started. An ordering of nodes / `Pipers` of a graph / `Dagger` which has this property is called a postorder. There are possibly more then one postorder per graph `Dagger`. The exact postorder used to connect the `Pipers` has some additional properties

- all down-stream `Pipers` for a `Piper` (A) come before the next `Piper` (B) for which no such relationship can be established. This can be thought as maintaining branch contiguity.
- such branches can additionally be sorted according to the branch argument passed to the `Piper` constructor.

Another aspect of order of a `Dagger` is the sequence by which a down-stream `Piper` connects multiple up-stream `Pipers`. The inputs cannot be sorted based solely on their postorder because the down-stream `Piper` might be connected directly to a `Piper` to which one of it's other inputs has been connected before. The inputs of a `Piper` are additionally sorted so that all down-stream `Pipers` come before up-stream `Pipers`, while `Pipers` for which no such relation can be established are still sorted according to their index in the postorder. This can be thought of as sorting branches by their "generation".

You could think of a workflow as an `imap` function composed from nested `imap` functions i.e.:

```
# nested imaps as pipelines
pipeline = imap(h, izip([imap(f, input_for_f), imap(g, input_for_g)]))
```

This is a pipeline of 3 functions `f`, `g`, `h`. Functions `f` and `g` are upstream relative to `h`. Because of the `izip` function `input_for_f` and `input_for_g` have to be of the same lenght.

A started `Dagger` is able to process input data. The simplest way to process all inputs is to zip it's output `Pipers`:

```
output_pipers = dagger_instance.get_outputs()
final_results = zip(output_pipers)
```

If any of the `Pipers` used within a `Dagger` uses an `Numap` instance and the `Dagger` is started. The Python process can only be exited cleanly if the `Dagger` instance is stopped by calling its `stop` method.

1.4.10 The Plumber

The `Plumber` is an easy to use interface to **PaPy**. It inherits from the `Dagger` object and can be used like a `Dagger`, but the `Plumber` class adds methods related to the “run time” of a pipeline. A `Plumber` can start/run/pause/stop a pipeline and additionally load and save a workflow (not implemented) A **PaPy** workflow is loaded and saved as executable Python code, which has the same privileges as the Python process. Please keep this in mind starting workflows from untrusted sources!

1.4.11 The additional components

Those classes and functions are used by the core components, but are general and might find application in your code.

- `DictGraph` and `Node` - Two classes which implement a graph data-structure using a recursively nested dictionary. This allows for simplicity of algorithms/methods i.e. there are no edge objects because edges are the keys of the `Node` dictionary which in turn is the value in the dictionary for the arbitrary object in the `DictGraph` instance i.e.:

```
from papy import Graph
graph = Graph()
object1 = '1'
object2 = '2'
graph.add_edge(object1, object2)
node_for_object1 = graph[object1]
node_for_object2 = graph[object2]
```

The `Dagger` is a `DictGraph` object with directed edges only and no cycles.

- `imports` - a function wrapper, which allows to inject import statements to a functions local namespace at creation (code execution) e.g. on a remote Python process.

1.5 About the Parallelism in PaPy

This document explains and gives code snippets how to use the parallel features of *PaPy*. The first paragraphs introduce the `map` and `imap` functions and the *IMap* object. The types of parallelism, are explained in the later sections where also typical optimizations, bottlenecks of pipelines are covered.

Parallel functionality is provided by the *IMap* class. In the most basic mode of operation it can be called exactly like `itertools.imap` or the `multiprocessing.Pool` `imap` method. Setting additional options allows to parallelise the evaluation using *Workers* of threads, processes or remote processes and share those workers among multiple functions and inputs. A unique feature of *IMap* is that it allows to fine-tune the memory-consumption, parallelism and laziness trade-off of nested function maps.

The interpreter will hang on exit if a pipeline does not finish or is halted abnormally

The Python interpreter exits (returns) if all spawned threads or forked processes return. *PaPy* uses multiple threads to manage the pipeline and evaluates functions in separate threads or processes. All of them need to be stopped before the parent python process can return. This is done automatically whenever a pipeline finishes or some expected exception occurs, in all other cases it is required that the user stops the pipeline manually.

1.5.1 Map basics

A map function applies a function to the items of a sequence. This is in Python expressed with the following syntax:

```
def power(x):
    return x*x

inp_list = [1,2,3,4]
out_list = map(power, inp_list)
```

The resulting output is [1, 4, 9, 16]. Though unmatched in simplicity this function has several computational drawbacks.

1. results are evaluated sequentially i.e. `power(1)`, `power(2)`, `power(3)`, `power(4)` and on a single processor.
2. The function returns only after *all* results have been calculated.
3. The list of results has to fit into memory together with the input.
4. The results are always returned in order.

The last two issues can be addressed by using the `imap` (iterated map) function. Its usage is almost as simple:

```
from itertools import imap
inp_list = [1,2,3,4]
out_iterator = imap(power, inp_list)
first_result = out_iterator.next() # returns power(1)
second_result = out_iterator.next() # returns power(2)
```

There are however a number of differences between the two. The `imap` function returns a result object immediately, but this object is only a link (via the `out_iterator.next` method) to the next result to be calculated. The evaluation starts as soon as the next method is called (lazy evaluation) and the result is returned as soon as it is calculated. The function, argument tuples are still evaluated sequentially and on a single processor, but they are returned as soon as they are needed and only a single result needs to fit into memory.

Recent versions of Python (2.6+) provide implementations of a parallel map where results are evaluated by a pool of worker processes. This functionality comes in three flavours:

```
from multiprocessing import Pool
pool = Pool()
out_list = pool.map(power, inp_list)
out_iterator1 = pool.imap(power, inp_list)
out_iterator1.next(timeout =1)
out_iterator2 = pool.imap_unordered(power, inp_list)
```

Warning: This example like most of the following is a code snippet. It is or should be syntactically correct. But might outside of a valid script file.

The `Pool.map` does exactly the same as the simple `map` function, but it uses by default all available cores. This addresses the second drawback. Both `imap` methods return a result object with the next method like `itertools.imap`. Calling it returns the next calculated (`imap_unordered`) or expected result (`imap`). The next method has an optional `timeout` argument i.e. if no result is available within the limit a `TimeoutError` is raised. Although it might seem that those implementations have none of the above mentioned drawbacks several implementation choices make them inappropriate for constructing pipelines.

1.5.2 A pipeline is a nested `imap`

Map functions have a list as input and return a list so they can be nested i.e.:

```
from math import degrees, radians
map(degrees, map(radians, [1,2,3]))
[1.0, 2.0, 3.0]
```

In this example the function `radians` is first applied to the elements of the input list then the the results are back-converted to `degrees`. The order of evaluation:

```
temp_list = [] temp_list.append(radians(1)) temp_list.append(radians(2)) temp_list.append(radians(3))
result_list = [] result_list.append(degrees(temp_list[0])) result_list.append(degrees(temp_list[1])) re-
sult_list.append(degrees(temp_list[2])) << return list of results >>
```

If we use an iterated (lazy) version of the `map` function i.e.:

```
from itertools import imap
imap(degrees, map(radians, [1,2,3]))
[1.0, 2.0, 3.0]
```

The order of calculation changes:

```
temp_result = radians(1) result = degree(temp_result) <<return result>> temp_result = radians(2) result
= degree(temp_result) <<return result>> temp_result = radians(3) result = degree(temp_result) <<return
result>>
```

Note that only one temporary result needs to be stored at any given time and that the first result is returned after only two calculations. But multiprocessing Pools `imap` function yield yet another calculation order.:

```
from multiprocessing import Pool
pool = Pool()
pool.imap(degrees, pool.imap(radians, [1,2,3]))
[1.0, 2.0, 3.0]
```

What happens is a little bit unexpected, first the function is evaluated (by multiple processes) and only after all temporary results are calculated the second functions is iteratively applied:

```
temp_list = []
temp_list.append(radians(1))
temp_list.append(radians(2))
temp_list.append(radians(3))
result = degree(temp_result[0])
<<return result>>
result = degree(temp_result[1])
<<return result>>
result = degree(temp_result[2])
<<return result>>
```

The results are either returned immediately or stored in a result list. The maximum size of this list is the size of temporary list and the size of the input. The reason for this behaviour is the order by which tasks i.e. (function, data) tuples are submitted to the pool. If one pool handles two functions first all (radians, x) tuples are submitted and then all (degrees, x). The outer function is evaluated last so for multiple and computationally expensive functions the first result might be available after a long lag phase followed by a burst of results.

This problem can be solved by having a separate pool for each function.:

```
from multiprocessing import Pool
pool1 = Pool()
pool2 = Pool()
pool2.imap(degrees, pool1.imap(radians, [1,2,3]))
[1.0, 2.0, 3.0]
```

Now the execution order is not defined anymore as processes in the two Pools compete for CPU time from the OS. A possible evaluation order might be like this:

```
temp_list = []
temp_list.append(radians(1))
temp_list.append(radians(2))
result = degree(temp_result[0])
<<return result>>
result = degree(temp_result[1])
<<return result>>
temp_list.append(radians(3))
result = degree(temp_result[2])
<<return result>>
```

As you can see a temporary result list is still built. Its maximum length is not predictable and limited by the length of the input. Another drawback is that if the number of functions is big the number of process-pool workers significantly exceeds the number of available CPUs or CPU-cores, which is inefficient.

1.5.3 The task and the tasklet

The `imap` implementation in *PaPy* (*IMap*) is different as it allows to control the order by which function and data tuples are submitted to the worker pool. It introduces the concept of a task which is a function, input and arguments tuple. The input is a python iterator i.e. an object which has a `next` method it obviously should return data to be calculated next. The argument is a tuple of parameters which is given to the function for example:

```
(function, data_iterator, ('rome', 1.17, some_object)) # a task
```

The `IMap` role is to evaluate tasks. To evaluate a task means to evaluate all tasklets:

```
(function, data_iterator.next(), 'rome', 1.17, someobject) # a tasklet
result = function(data_iterator.next(), 'rome', 1.17, someobject) # evaluation
```

Until the `data_iterator` is empty.

1.5.4 IMaps parallelism is defined by a stride

`IMap` allows to control the order in which tasklets are evaluated. This is accomplished by the `stride` parameter. A stride is the number of tasklets from one task submitted before any tasklet from the next task. The default stride is equal to the number of pool workers and should not be smaller.:

```
from IMap import IMap
from math import radians, degrees
Imap = IMap(worker_num =2)
output = Imap.add_task(radians, [1,2,3,4])
result = Imap.add_task(degrees, output)
Imap.start() # finished adding tasks
result.next() # or Imap.next(task =1) 0 is the first task
```

In this example the `Imap` instance has a pool with two workers (by default those workers are separate processes), its default stride is therefore 2. The order in which the tasks will be evaluated is as follows.:

```
temp_list = []
result_list = []
temp_list.append(radians(1))
temp_list.append(radians(2))
result_list.append(degree(temp_list[0]))
<<return result>>
result_list.append(degree(temp_list[1]))
<<return result>>
```

```

temp_list = []
result_list = []
temp_list.append(radians(3))
temp_list.append(radians(4))
result_list.append(degree(temp_list[2]))
<<return result>>
result_list.append(degree(temp_list[3]))
<<return result>>

```

The `temp_list` (in fact it is a queue) has a defined size limit (stride) and so is the `result_list`. The details of memory consumption will be explained in the next paragraphs here it suffices to say that a minimum memory requirement of 2 temporary results can be enforced on this pipeline without loss of efficiency. If the pipeline was longer and had computationally expensive functions it would be noticeable that the results from the outer function arrive in burst of 2 or bursts of stride size.

1.5.5 IMap needs tasks in the right order

In the previous example two nested tasks have been added to the IMap function using the `add_task` method. The general way of working with IMap is as follows:

```

#0. import IMap
from IMap import IMap
#1. define imap keyworded parameters e.g.
imap_instance = IMap(worker_remote = [['host', 2]])
#2. add tasks
out0 = imap_instance.add_task(function0, input_data)
out1 = imap_instance.add_task(function1, out0)
out2 = imap_instance.add_task(function2, other_data)
#3. start the evaluation
imap_instance.start()
#4.
<< get the results >>

```

In this section we will focus on step 2. We have submitted 3 tasks to the `imap` instance. Because the order of submission matters they will be evaluated in the order.:

```

# first stride
function0(input_data[0 .. n]) # where n is stride
function1(out1[0 .. n])
function2(other_data[0 .. n])

# second stride
function0(input_data[n .. n+n]) # where n is stride
function1(out1[n .. n+n])
function2(other_data[n .. n+n])1

# and so on

```

Because `function1` depends on the results from `function0` it can't be added as a task before `function0`. It might seem impossible because `function1` takes the output of `function0` (`out0`) as an argument, but in general the input could be an object created before `out0`, which is modified with `out0` after creation. This is possible because evaluation starts only after the `start` method is called.

1.5.6 IMap can limit the memory consumption.

By default the maximum memory consumption of an IMap instance is equal to the number of tasks times the stride size, but this limit can be changed. Consider an imap instance with two tasks, which are not nested and a stride of 3, this means that the default maximum memory consumption is 6. The evaluation will pause whenever the IMap instance reaches the limit. In pseudo-python:

```
list0 = [] list1 = [] # a stride of 3 list0.append(function0(arg0)) list0.append(function0(arg1))
list0.append(function0(arg2)) # list0 has size 3 list1.append(function1(arg0))
list1.append(function1(arg1)) list1.append(function1(arg2)) # list1 has size 3
```

Because at this moment the two list have together a length of 6 no further evaluations takes place. The only way to clear a list is to get results from the output iterators (say out0 for function0 and out1 for function1). If we take a single result say:

```
result_0_0 = out0.next() # submits function0(arg3) to the pool
```

memory consumption lowers to 5 and the next task is submitted to the pool.:

```
list0.append(function0(arg3))
```

memory consumption is once again at 6 and the next task (function0, arg4) waits. By retrieving results from the output iterators we free the temporary result lists (queues) and allow evaluation to proceed. Results do not have to be retrieved in the order the tasks have been submitted to the pool or the order in which the results have been calculated. Assume that in the last example the next method of out0.next() has been called 5 more times:

```
result_0_1 = out0.next() # submits function0(arg4) to the pool
result_0_2 = out0.next() # submits function0(arg5) to the pool
result_0_3 = out0.next() # submits function1(arg3) to the pool !note 1
result_0_4 = out0.next() # submits function1(arg4) to the pool !note 1
result_0_5 = out0.next() # submits function1(arg5) to the pool !note 1
```

after all workers finish list0 the imap reaches a stage where:

- list0 - will be empty
- list1 - will have 6 results (arg0 - arg5)
- task (function0, arg6) will wait to be submitted

List0 is empty and the next task (function0, arg6) cannot be submitted because the total memory consumption is 6. If we would call out0.next the result would never arrive and the python interpreter would be blocked. A timeout argument can be supplied it causes the next method to raise a TimeoutError if after the specified number of seconds no result is available.:

```
result_0_6 = out0.next(timeout =2) # raise after 2 seconds
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
multiprocessing.TimeoutError
```

We have to empty the other functions output (out1) to get the 7th result for out0. Note that the order of task submissions is defined at start by how the memory is freed (the order in which out0.next nad out1.next are called) does not change this order:

```
result_1_0 = out1.next() # submits function0(arg6) to the pool
result_0_6 = out0.next() # submits function0(arg7) to the pool
```

To never run into an block only it is necessary to:

- retrieve at most stride number of results from any output in a sequence
- retrieve the result n from outputN before the result n from outputN+1

The most memory efficient way to do this is to get the results in batches of stride size, which is equivalent to the order the tasks have been submitted to the worker pool. If the two functions from the above example were nested this would happen automatically for the inner function. If the the pipeline run in the most memory efficient way the memory consumption can be lowered to the stride of the IMap this is done using the buffer argument. For example:

```
Imap = IMap(stride =3, buffer=5)
```

1.5.7 Parallel: local vs. remote and threads vs. processes

IMap supports parallelization using local threads (`worker_type='thread'`) and processes (`worker_type='process'`). Remote threads and processes are run by local processes. IMap is designed to allow the user to choose the type of parallelization using the `worker_type` argument and/or by specifying the remote processes using the `worker_remote` argument.

Because of the global interpreter lock (GIL) in the standard cPython implementation of the Python programming language, only one os-thread can execute interpreted python code. Multiple running threads are assigned timeslices of “code access”. In general it is not possible to speed cpu-bound computations using threads. Multiple threads can however speed up certain functions to a certain degree if either the function is:

- IO-bound (e.g. waits for server responses)
- uses libraries which release the global interpreter lock

The first case applies to all function which depend on user interaction and other blocking Input/Output operations like reading or writing a file. the second case applies mostly to external compiled libraries doing cpu-intensive calculations. Those libraries can natively be multithreaded. An IMap running tasks in worker threads is therefore a good choice if the tasks submitted to the function are IO-bound or if it uses a library which releases the GIL for significant periods of time.

The GIL can be circumvented by forking the python interpreter process instead of spawning additional threads within. Such forked processes have separate memory space and are seen by the operating system as another python interpreter. Multiple processes have each their own GIL and are therefore suited to parallelize interpreted python code. This parallelization will make the computation faster if the operating system has enough resources to support the processes. If a function is CPU-bound the most limiting resource is the CPU and therefore the number of processes should not exceed the number of available CPUs. Using multiple processes within on parent process is called multiprocessing and python 2.6 and above support this feature via the multiprocessing module out of the box on most operating systems (Linux, MacOSX, Windows). There are however implementation differences among UNIX systems and Windows which make multiprocessing on Windows less efficient. This module is also back-ported to python 2.5 as external module. It is also possible to parallelize local computations using the RPyC library. For cpu-bound tasks the additional overhead is minor.

If a single machine is not fast enough for the task, distributed computing i.e. computation on remote physical computers might be considered. IMap supports distributed computation using the RPyC library. To use this feature a “classic” RPyC server has to be running on a remote host. This server allows clients (i.e. IMap instances) to connect to them and execute functions. An RPyC server can be both thread or process based. Only a process-based RPyC will be able to use multiple CPUs on the remote computer (not supported on Windows). An IMap which connects to remote servers spawns a new process for each thread/process spawned/forked remotely (by the RPyC server).

Note: IMap has no possibility to change the thread/process nature of the remote server.

In the following example snippet:

```
Imap = IMap(worker_type ='process', worker_num =2, worker_remote =[['host1',1], ['host2', 4]])
```

An IMap instance is created which uses worker processes. It has a total of 7 local worker processes. Five of the local worker processes do the computation on remote hosts: 4 on host2 and 1 on host1.

The following is illegal, because remote threads/processes can be managed only by ‘process’ workers:

```
Imap = IMap(worker_type='process', worker_num=2, worker_remote=[[ 'host1', 1], [ 'host2', 4]])
```

The RPyC server can also be started on the local machine ('localhost'). In this example the IMap instance has a total of 2 local worker processes which manage two remote processes which happen to exist on the same physical machine.:

```
Imap = IMap(worker_type='process', worker_num=0, worker_remote=[[localhost', 2]])
```

1.5.8 If the order of the results is not important

IMap supports unordered results via the `unordered` argument:

```
Imap = IMap(ordered=True) # the default is ordered
Imap = IMap(ordered=False) # random order of results
```

If the results are allowed to be unordered the evaluation might be significantly faster under certain circumstances described later, but this order is not reproducible. As a general advice do not use unordered Imap instances in branched papy pipelines, unless you really know what you are doing.

1.5.9 Timeouts and skipping

IMap supports timeouts. A `TimeoutError` (from the multiprocessing module) is raised whenever a result for a given task cannot be returned within approximately the number of seconds specified. The timeout is only approximate because IMap uses multiple threads to manage the input and output queues for the worker threads/process. The thread, which receives result might not have access to the interpreter when the timeout passes.

The skipping argument allows to skip results which did timeout. If skipping is not specified IMap will try to return the same result (for this task) once more. If the timeout is not specified skipping is ignored. If IMap is used with nested tasks a timeout should in practice not be specified unless IMap is used within from a piper object or the IMap instance and timeouts are specified the skipping argument should be true, the reason for this is that

1.5.10 The parallel stride revisited

In one of the previous sections it has been described how IMap allows for parallelism by introducing the concept of the stride. To recapitulate a stride is the number of tasklets submitted to the pool for a specific task to be executed/evaluated in parallel. Another tasklet can be submitted if the buffer is larger than the stride or as soon as any of the results of the parallel evaluations is retrieved. The user can add multiple tasks to one IMap instance. If the evaluation is CPU-bound and the IMap uses worker processes an optimal speed up equal to the number of CPUs or CPU-cores. However because of inter task dependencies i.e. nested functions, the speed-up might be smaller as a result of the memory trade-off and task submission order.

#. The output needs to be retrieved. If the output of an IMap instance is not retrieved it will pause whenever it fill the buffer of temporary results, therefore it is important to retrieve the results as soon as they are ready. Because results generally arrive in batches of stride size it is best to try to retrieve stride number of results from each task. If the IMap is used within a papy pipeline the Plumber has a separate thread (started using the `plunge` method) dedicated to keep the IMap evaluating.

1. Variable tasklet calculation times

If the next task depends on the results from the first task then it's first tasklet

1.6 Inter Process Communication

This chapter deals with the details of inter-process communication (IPC) in **PaPy**. By its design *PaPy* is a rather high-level engine for the execution of workflows and the details of inter-process communication are by default hidden from the user. This does not mean that it is not possible to influence how processes are communicated i.e. synchronize and exchange data. This allows for optimization of workflow execution times, but requires some understanding of the involved concepts and is done at the cost of generality e.g. knowing that two processes will execute on a shared memory UNIX system allows to communicate them via pipes (FIFOs), which will skip some computation and computations. This section should be read together with the API documentation for `papy.util.func.dump_item` and `papy.util.func.load_item`. By default in **PaPy** interprocess communication happens between `Pipers` and the manager process.

1.6.1 When does it happen?

A `Piper` object is assigned to an `NuMap` instance which uses a process/thread pool to parallelize the evaluation. All functions used to create a `Worker` are evaluated in a single call and no IPC is necessary. Two `Pipers` on the other hand might be executed in different processes and on different machines running possibly different operating systems. Therefore by default `Pipers` are connected by a **pair** of locked `pipe` objects via a manager process, which is the process used to execute/start the workflow.

1.6.2 Why is it inefficient?

Using a manager process to communicate two other process is inefficient as it involves two passes of pickling/unpickling and creates a potential bottleneck because half of all serialization computation is done by a single intermediate process, this obviously will not scale if the number of process to be communicated is large. The solutions are a) bypass the double pipe connection and connect processes directly, b) use a more efficient serialization protocol, c) eliminate IPC by collapsing multiple `Piper` instances into a single one. We will focus here on the first option, which involves adding dumping(output) and loading(input) worker-functions to the `Workers` instances i.e. in pseud-code:

```
from papy import workers
upstream = Worker((func, workers.io.pickle_dumps, workers.io.dump_item),\
                  (), (), ('tcp'))
downstream = Worker((workers.io.load_item, workers.io.pickle_loads, func),\
                    (), (), ())
up_piper = Piper(upstream, parallel =some_IGMap_instance)
down_piper = Piper(downstream, parallel =some_different_IGMap_instance)
pipes = Plumber()
pipes.add_pipe((up_piper, down_piper))
```

In this example we created two `Worker` instances, which are used to create `Piper` instances connected within a `Dagger` instance and executed by different processes possibly on different physical machines. Because of this a networked method of communication has been chosen 'tcp'. This method involves sending data over a network socket. Data has to be serialized before it can be passed to another process. Within `multiprocessing` this is done via pickling, `RPyC` uses an internal protocol called brine. *PaPy* has built-in workers which support the pickle, json and marshall protocols. Pickle is the most general protocols and most Python objects can be pickled. Currently Json and marshall might be faster, but they have limitations (compability between Python versions and the range of serializable Python objects). The `load_item` worker will auto discover the type of communication. The currently supported methods of communication are 'tcp' or 'udp' for network communications, 'fifo' or 'shm' to communicate processes on the same physical machine and hooks to databases currently 'sqlite' and 'mysql'. Data can also be exchanged via temporary files 'file'. Temporary files can be used to communicate remote hosts if they are accessible from both e.g. via NFS or Samba.

How does it work?

If the user decides to use custom communication methods the inefficient double-pipe connection is used to transfer only a very small amount of data and effectively to synchronize the processes. For example for TCP based communication it is the hostname and port and the type of the protocol ‘tcp’. For file, shm and FIFO based communication it is just the file name. This amount of data will not be a bottleneck for pipelines of any size.

How do database “hooks” work?

The *PaPy* `dump_db_item` and `load_db_item` worker functions allow to communicate Python processes via a database. The data in the database can be stored only until it is retrieved or persistantly and serve as a way to check-point the pipeline. How the data is stored in a database depends on the type of the database, currently ‘mysql’ and ‘sqlite’ databases are supported. Sqlite database files should not be shared over NFS, but can be written and read by different processes.:

```
from papy import workers
upstream = Worker((func, workers.io.pickle_dumps, workers.io.dump_db_item),\
                  (), (), ('sqlite'))
downstram = Worker((workers.io.load_db_item, workers.io.pickle_loads, func),\
                   (), (), ())
up_piper = Piper(upstream, parallel =some_Imap_instance)
down_piper = Piper(downstram, parallel =some_different_Imap_instance)
pipes = Plumber()
pipes.add_pipe((up_piper, down_piper))
```

Which method should I choose?

The recommended method of communication depends whether the processes run on the same or different physical machines, what the operating systems of the Python processes and the size of the exchanged data. Small lists, objects, strings, etc. (the size of up to hundreds of kB after serialization) should be transferred using the default method i.e. without using the `dump_item` worker. Only for large objects non-standard communication methods should be considered. If the processes run on a shared memory UNIX system you should use FIFOs (pipes) or shared memory (this requires the `posix_ipc` module). Using FIFOs on Windows systems is currently not supported (but might be in future) and Windows is not POSIX compliant so you are left with files, which might be fast enough for typical applications. Files on a network share are also the recommended method to communicate Windows-based processes. Networking i.e. ‘tcp’ and ‘udp’ requires forking of the process, which evaluates the `dump_item` function. Forking is not supported on Windows, but should work well on all UNIX systems. UDP should provide higher performance than TCP, but should only be used on reliable, collision-less networks. When using UDP you are not guaranteed that all data will be transmitted over the network, this will yield `WorkerErrors`, which in turn will require you to re-run the pipeline for failed input items.

1.7 The Produce / Spawn / Consume Idiom

This section introduces the Produce / Spawn / Consume idiom that allows e.g. to evaluate a single data item multiple times (e.g. for different parameters)

1.8 Runtime (Logging, Interaction, Errors and Timeouts)

A started workflow evaluates the functions of the processing nodes on the data items traversing the pipeline using the assigned computational resources. The execution happens in the background i.e. the user has the choic to interact with

the running pipeline *via* the `Plumber` object. Further components of a workflow i.e. `Piper` and `Numap` instances log runtime messages. These logs can be saved to disk and inspected when needed.

Logging behavior is customized by `papy.util.runtime`, interaction is possible through `papy.core.Plumber`, exceptions, error and timeouts are handled by the processing nodes i.e. `papy.core.Piper` instances.

1.8.1 Logging

A function to configure at what importance level and how logged messages should be saved or displayed is available in the `papy.util.runtime` module.

1.8.2 Interaction with the Plumber

Please see the methods of the `Plumber` object.

1.8.3 Errors and Exceptions

PaPy workflows are by default resistant to all exceptions that occur as a result of error in user-provided worker functions.

1.8.4 Dealing with Timeouts

To Be Written.

1.9 Optimization

The throughput of a pipeline will be most significantly limited by the slowest `Piper`. A processing node might be slow either because it does a CPU-intensive or IO-intensive task, because it waits for some data, or because it synchronizes with other nodes and waits.

1.9.1 Identifying bottlenecks

As a general rule you should optimize the bottleneck(s) only. Therefore it is critical to understand where and what the bottleneck is.

This has good reason as most of your nodes will not limit the throughput of the workflow while parallelization is quite expensive. If your pipeline has no obvious bottleneck it's probably fast enough. If not you might be able to use a shared pool.

1.9.2 Understanding bottlenecks

To Be Written.

Addressing synchronization

1.9.3 Unordered Pipers

Unordered pipers return results in an arbitrary order e.g for the input sequence `[3, 2, 1]` a parallel unordered `Piper` instance with a function that doubles the input might return `[6, 2, 4]` or any other permutation of the doubled numbers. Unordered nodes do not compute faster they only make the results available sooner. Thus a down-stream computation that uses the same computational resource can start earlier and potentially utilize it to a fuller extent. You should consider unordered `Pipers` if the computation time for data items varies significantly.

Addressing serialization

To Be Written.

Distributing Computational resources

As a general rule of you most likely should not use a shared `Numap` instance among all `Pipers` within a workflow.

If the throughput of your pipeline is limited by a cpu-intensive tasks you should parallelize this node. **PaPy** allows to parallelize cpu-bound `Pipers`. The amount of cpu-power should be proportional to the computational requirements of a processing task. The number of recommended `Numap` pool worker processes should equal or slightly larger than the number of physical CPU-cores on each local or remote computer.

1.10 Examples

Examples can be found in the `doc/examples` directory of the source distribution. See also: `test/test_core.py` and `test/test_util_func.py` to explore the API. To learn about `Numap` see the documentation for the “numap” package.

1.11 PaPy for Bioinformatics

PaPy makes it easier to create bioinformatics workflows in a semi-standardized manner by using data-containers for biological data. These containers are specialized sub-classes of typed `NuBoxes` and are available `nubio` package, please refer to the `NuBio` documentation to learn about bioinformatics data-containers utility functions and to the `nubox` documentation for the hierarchical `NuBox` data-structure.

1.12 Workflows

Complete workflows can be found in `doc/workflows`.

1.13 Known Issues

1.13.1 PaPy in the Interactive Interpreter

Parallel features of **PaPy** do **not** work in the interactive interpreter. This is a limitation of the Python `multiprocessing` module.

This means that **PaPy** workflows can be created, manipulated, tested and run from within the interactive interpreter freely as long as they do not use the parallel or remote evaluation.

Code snippets, examples and use cases are not meant to be typed into the interactive interpreter console. They should be run from the command line:

```
$ python example_file.py
```

The reason for this is that functions defined in the interactive interpreter will not work (and will hang Python) if passed into `NuMap` instances! A Python function can only be communicated to multiple processes if it can be serialized i.e. pickled. This is not possible if the function is in the same namespace as a child process (created using the multiprocessing library).

1.13.2 Object Picklability

Objects are submitted to the worker-threads by means of queues to worker-processes by pipes and to remote processes using sockets. This requires serialization, which is internally done by the `cPickle` module. Additionally `RPyC` uses it's own 'brine' for serialization. The submitted objects include the functions, data, arguments and keyworded arguments all of which need to be picklable!

Worker-methods are impossible

Class instances (i.e. Worker instances) are picklable only if all of their attributes are. On the other hand class instance methods are not picklable (because they are not top-level module functions) therefore class methods (either static, class, bound or unbound) will not work for a parallel piper.

File-handles make remotely no sense

Function arguments should be picklable, but file-handles are not. It is recommended that output pipers store data persistently, therefore output workers should be run locally and not use a parallel `IMap`, circumventing the requirement for picklable attributes.

1.14 FAQ

1.15 Dictionary of terms and definitions

A dictionary of terms used within the documentation.

1.15.1 map

Higher-order map function. A function which evaluates another function on all elements of the input collection.

1.15.2 imap

Iterated higher-order map function. A function which evaluates another function on all elements of the input collection returning and evaluating the results iteratively and lazily. **PaPy** depends on the `imap` implementation provided by the standard Python `imap` from `itertools.imap` and the advanced `NuMap`.

1.15.3 NuMap

A parallel implementation of a multi-task `imap` function, which is used within **PaPy**. It uses a pool of worker-threads or worker-processes and evaluates functions in parallel either locally or remotely.

1.15.4 Worker function

A function with a standardized input written to be used by a `Worker` class instance. All processing of a **PaPy** pipeline has to be coded as `Worker` functions.

1.15.5 worker process / thread

A thread or process inside an `NuMap` instance evaluating a tasklet remotely or locally.

1.15.6 Worker

An object-oriented wrapper for worker functions, it is roughly equivalent to a “function with partially applied arguments”.

1.15.7 Piper

An object oriented wrapper for `Worker` instances, corresponds to “worker with defined mode of evaluation”.

1.15.8 Dagger

An directed acyclic graph (DAG) to store and connect `Piper` instances.

1.15.9 Plumber

A wrapper for the `Dagger` designed to run and interact with a running pipeline.

1.15.10 input stream

The input stream is the data that enters a **PaPy** pipeline. The data is assumed to be a collection of items expressed as a Python iterator (or any object which has the `next` method).

Any sequence (e.g. a list or a tuple) can be made into an iterator using the Python built-in `iter` function e.g:

```
sample_sequence = [data_point1, data_point2, data_point3]
sample_iterator = iter(sample_sequence)
```

Files are by default line-iterators i.e.:

```
sample_file = open('sample_file.txt')
sample_file.next() # returns the first line
sample_file.next() # returns the second line
```

1.15.11 output stream

Input item saved (to disk) by an output `Piper`. By default the output `Piper` should return a `None` for every input item, but save the result persistently (somehow/somewhere).

1.15.12 item

A single element of the data stream.

1.15.13 input Piper

A `Piper`, which is connected to a input stream (or multiple input streams) is a input `Piper`. Such a `Piper` corresponds to a node in the graph which has no upstream nodes within the **PaPy** workflow or in other words has no outgoing edges in the directed acyclic graph. An input `Piper` is an input node in the graph representing the workflow.

1.15.14 output Piper

A `Piper`, which generates the output stream is an output `Piper`. A **PaPy** workflow might have multiple output `Pipers` in different places of the pipeline. An output `Piper` corresponds to a node in the graph which has no downstream nodes within the pipeline or in other words has no incoming edges in the directed acyclic graph. An output `Piper` is an output node in the graph representing the pipeline.

1.15.15 lazy evaluation

Is the technique of delaying a computation until the result is required.

1.15.16 task

A task is an ordered `tuple` of objects added to the `Numap` instance it consists of:

- a function, which will be evaluated on the input element-wise
- an input (a `list`, `tuple` or any iterator object like an `array`)
- a `tuple` of arguments e.g. (`arg1`, `arg2`, `arg3`)
- a `dict` of keyword arguments i.e. `{'arg1': value_1, 'arg2': value_2}`

The optional arguments and keyworded arguments have to match the signature of the function. The task is iteratively split into evaluated calls in the following way:

```
(func, element_from_iterable, arguments, keyworded_arguments)
result = func(element_from_iterable, arguments, keyworded_arguments)
```

1.15.17 inbox

The first argument of any `Worker` function. The elements of the function correspond to the outputs of the upstream function in the `Worker` instance or to outputs of other `Pipers`. These outputs are defined by the pipeline topology. The contents of the inbox depend on a specific input item to the pipeline. All other arguments of a worker function are predetermined.

1.16 PaPy API

This part of the documentation is generated automatically from the source code documentation strings. It should be the most up-to-date version. If there is a conflict between the hand-written and and generated documentation, please contact the author e.g. by adding an issue on the project page.

1.16.1 `papy.core`

This module provides classes and functions to construct and run a **PaPy** pipeline.

class `papy.core.Dagger` (*pipers=()*, *pipes=()*, *xtras=None*)

The `Dagger` is a directed acyclic graph. It defines the topology of a `PaPy` pipeline / workflow. It is a subclass of `DictGraph`. `DictGraph` edges are called within the `Dagger` `pipes` and have an inverted direction which reflects dataflow not dependency. Edges can be thought of as dependencies, while `pipes` as dataflow between `Pipers` or nodes of the graph.

Arguments:

- `pipers(sequence)` [default: `()`] A sequence of valid `add_piper` inputs (see the documentation for the `add_piper` method).
- `pipes(sequence)` [default: `()`] A sequence of valid `add_pipe` inputs (see the documentation for the `add_piper` method).

add_pipe (*pipe*, *branch=None*)

Adds a pipe (A, ..., N) which is an N-tuple tuple of `Pipers` instances. Adding a pipe means to add all the `Pipers` and connect them in the specified left to right order.

The direction of the edges in the `DictGraph` is reversed compared to the left to right data-flow in a pipe.

Arguments:

- `pipe(sequence)` N-tuple of `Piper` instances or objects which are valid `add_piper` arguments.
See: `Dagger.add_piper` and `Dagger.resolve`.

add_piper (*piper*, *xtra=None*, *create=True*, *branch=None*)

Adds a `Piper` instance to this `Dagger`, but only if the `Piper` is not already there. Optionally creates a new `Piper` if the “`piper`” argument is valid for the `Piper` constructor. Returns a tuple (`new_piper_created`, `piper_instance`) indicating whether a new `Piper` has been created and the instance of the added `Piper`. Optionally takes “`branch`” and “`xtra`” arguments for the topological node in the graph.

Arguments:

- `piper(Piper, Worker or id(Piper))` `Piper` instance or object which will be converted to a `Piper` instance.
- `create(bool)` [default: `True`] Should a new `Piper` be created if “`piper`” cannot be resolved in this `Dagger`?
- `xtra(dict)` [default: `None`] Dictionary of `graph.Node` properties.

add_pipers (*pipers*, **args*, ***kwargs*)

Adds a sequence of `Pipers` instances to the `Dagger` in the specified order. Takes optional arguments for `Dagger.add_piper`.

Arguments:

- `pipers(sequence of valid add_piper arguments)` Sequence of `Pipers` or valid `Dagger.add_piper` arguments to be added to the `Dagger` in the left to right order.

add_pipes (*pipes, *args, **kwargs*)

Adds a sequence of pipes to the Dagger in the specified order. Takes optional arguments for `Dagger.add_pipe`.

Arguments:

- `pipes`(sequence of valid `add_pipe` arguments) Sequence of pipes or other valid `Dagger.add_pipe` arguments to be added to the Dagger in the left to right order.

children_after_parents (*piper1, piper2*)

Custom compare function. Returns 1 if the first `Piper` instance is upstream of the second `Piper` instance, -1 if the first `Piper` is downstream of the second `Piper` and 0 if the two `Pipers` are independent.

Arguments:

- `piper1`(`Piper`) `Piper` instance.
- `piper2`(`Piper`) `Piper` instance.

connect (*datas=None*)

Connects `Pipers` in the order input -> output. See `Piper.connect`. According to the pipes (topology). If “datas” is given will connect the input `Pipers` to the input data see: `Dagger.connect_inputs`.

Argumentensts:

- `datas`(sequence) [default: None] valid sequence of input data. see: `Dagger.connect_inputs`.

connect_inputs (*datas*)

Connects input `Pipers` to “datas” input data in the correct order determined, by the `Piper.ornament` attribute and the `Dagger._cmp` function.

It is assumed that the input data is in the form of an iterator and that all inputs have the same number of input items. A pipeline will **deadlock** otherwise.

Arguments:

- `datas` (sequence of sequences) An ordered sequence of inputs for all input `Pipers`.

del_pipe (*pipe, forced=False*)

Deletes a pipe (A, ..., N) which is an N-tuple of `Piper` instances. Deleting a pipe means to delete all the connections between `Pipers` and to delete all the `Pipers`. If “forced” is `False` only `Pipers` which are not used anymore (i.e. have not downstream `Pipers`) are deleted.

The direction of the edges in the `DictGraph` is reversed compared to the left to right data-flow in a pipe.

Arguments:

- `pipe`(sequence) N-tuple of `Piper` instances or objects which can be resolved in the `Dagger` (see: `Dagger.resolve`). The `Pipers` are removed in the order from right to left.
- `forced`(bool) [default: `False`] The forced argument will be given to the `Dagger.del_piper` method. If “forced” is `False` only `Pipers` with no outgoing pipes will be deleted.

del_piper (*piper, forced=False*)

Removes a `Piper` from the `Dagger` instance.

Arguments:

- `piper`(`Piper` or `id(Piper)`) `Piper` instance or `Piper` instance id.
- `forced`(bool) [default: `False`] If “forced” is `True`, will not raise a `DaggerError` if the `Piper` has outgoing pipes and will also remove it.

del_pipers (*pipers, *args, **kwargs*)

Deletes a sequence of `Pipers` instances from the `Dagger` in the reverse of the specified order. Takes optional arguments for `Dagger.del_piper`.

Arguments:

- `pipers` (sequence of valid `del_piper` arguments) Sequence of `Pipers` or valid `Dagger.del_piper` arguments to be removed from the `Dagger` in the right to left order.

del_pipes (*pipes, *args, **kwargs*)

Deletes a sequence of pipes from the `Dagger` in the specified order. Takes optional arguments for `Dagger.del_pipe`.

Arguments:

- `pipes`(sequence of valid `del_pipe` arguments) Sequence of pipes or other valid `Dagger.del_pipe` arguments to be removed from the `Dagger` in the left to right order.

disconnect (*forced=False*)

Given the pipeline topology disconnects `Pipers` in the order `output -> input`. This also disconnects inputs. See `Dagger.connect`, `Piper.connect` and `Piper.disconnect`. If “forced” is `True` `Numap` instances will be emptied.

Arguments:

- `forced`(bool) [default: `False`] If set `True` all tasks from all `Numaps` instances used in the `Dagger` will be removed even if they did not belong to this `Dagger`.

get_inputs ()

Returns `Piper` instances, which are inputs to the pipeline i.e. have no incoming pipes (outgoing dependency edges).

get_outputs ()

Returns `Piper` instances, which are outputs to the pipeline i.e. have no outgoing pipes (incoming dependency edges).

resolve (*piper, forgive=False*)

Given a `Piper` instance or the `id` of the `Piper`. Returns the `Piper` instance if it can be resolved else raises a `DaggerError` or returns `False` depending on the “forgive” argument.

Arguments:

- `piper`(`Piper` or `id(Piper)`) a `Piper` instance or its `id` to be found in the `Dagger`.
- `forgive`(bool) [default: `False`] If “forgive” is `False` a `DaggerError` is raised whenever a `Piper` cannot be resolved in the `Dagger`. If “forgive” is `True` then `False` is returned.

start ()

Given the pipeline topology starts `Pipers` in the order `input -> output`. See `Piper.start`. `Pipers` instances are started in two stages, which allows them to share `Numaps`.

stop ()

Stops the `Pipers` according to pipeline topology.

exception `papy.core.DaggerError`

Exceptions raised or related to `Dagger` instances.

class `papy.core.Piper` (*worker, parallel=False, consume=1, produce=1, spawn=1, timeout=None, branch=None, debug=False, name=None, track=False, repeat=False*)

Creates a new `Piper` instance. The `Piper` is an object that acts a processing node in a PaPy pipeline.

A `Piper` can be created from a `Worker` instance another `Piper` instance or a sequence of functions or `Worker` instances in every case a new `Piper` instance is created.

Piper instances evaluate functions in parallel if they are created with a NuMap instance provided otherwise they use the `itertools.imap` function.

The “produce” and “consume” arguments allow for different than 1:1 mappings between the number of input and output items, while “spawn” allows accomodate a Piper to handle additional outputs. Additional outputs are created from the elements of the sequence returned by the wrapped Worker instance.

The product of “produce” and “spawn” of the upstream Piper has to equal the product of “consume” and “spawn” of the downstream Piper, for **each** pair of pipers connected.

The “branch” argument sets the “branch” attribute of a Piper instance. If two Pipers have no upstream->downstream relation they will be sorted according to their “branch” attributes. If neither of them has a “branch” attribute or both are identical their sort order will be semi-random. Pipers will implicitly inherit the “branch” of an up-stream Piper, thus it is only necessary to sepcify the branch of a Piper if it is the first one after a branch point.

It is possible to construct pipelines without specifying branches if Pipers which are connected to multiple up-stream Pipers (the order of which is by default semi-random) use Workers that act correctly regardless of the order of results in their inbox.

If “debug” is True exceptions are raised on all errors. This will most likely hang the Python interpreter after the error occurs. Use during development only!

Arguments:

- `worker(Worker, Piper or sequence of functions or ‘Workers’)`
- `parallel(False or NuMap)` [default: False] If parallel is False the Piper instance will not process data-items in parallel
- `consume(int)` [default: 1] The number of input items consumed from **all** directly connected upstream Pipers per one evaluation.
- `produce(int)` [default: 1] The number of results to generate for each evaluation result.
- `spawn(int)` [default: 1] The number of times this Piper is implicitly added to the pipeline to consume the specified number of results.
- `timeout(int)` [default: None] Time to wait till a result is available. Otherwise a PiperError is **returned** not raised.
- `branch(object)` [default: None] This affects the order of Pipers in the Dagger. Piper instances are sorted according to the data-flow upstream->downstream and their “branch” attributes. The argument can be any object which can be used by the `cmp` built-in function. If necessary they can override the `__cmp__` method.
- `debug(bool)` [default: False] Verbose debugging mode. Raises a PiperError on WorkerErrors.
- `name(str)` [default: None] A string to identify the Piper.
- `track(bool)` [default: False] If True results of this Piper will be tracked by the NuMap (ignored if Piper is linear).
- `repeat(bool)` [default: False] If True and “produce” is larger than 1 the evaluated results will be repeated. If False it assumes that the evaluated results are sequences and produce will iterate over that list or tuple.

connect (*inbox*)

Connects the Piper instance to its upstream Pipers that should be given as a sequence. This connects this Piper.inbox with the upstream Piper.outbox respecting any “consume”, “spawn” and “produce” arguments.

Arguments:

- `inbox(sequence)` sequence of `Piper` instances.

disconnect (*forced=False*)

Disconnects the `Piper` instance from its upstream `Pipers` or input data if the `Piper` is the input node of a pipeline.

Arguments:

- `forced(bool)` [default: `False`] If `True` the `Piper` will try to forcefully remove all tasks (including the spawned ones) from the `NuMap` instance.

next ()

Returns the next result. If no result is available within the specified (during construction) “timeout” then a `PiperError` which wraps a `TimeoutError` is **returned**.

If the result is a `WorkerError` it is also wrapped in a `PiperError` and is returned or raised if “debug” mode was specified at initialization. If the result is a `PiperError` it is propagated.

start (*stages=None*)

Makes the `Piper` ready to return results. This involves starting the the provided `NuMap` instance. If multiple `Pipers` share a `NuMap` instance the order in which these `Pipers` are started is important. The valid order is upstream before downstream. The `NuMap` instance can only be started once, but the process can be done in 2 stages. This methods “stages” argument is a `tuple` which can contain any the numbers 0 and/or 1 and/or 2 specifying which stage of the start routine should be carried out:

- stage 0 - creates the needed `itertools.tee` objects.
- stage 1 - activates `NuMap` pool. A call to `next` will block..
- stage 2 - activates `NuMap` pool managers.

If this `Piper` shares a `NuMap` with other `Pipers` the proper way to start them is to start them in a valid postorder with stages (0, 1) and (2,) separately.

Arguments:

- `stages(tuple)` [default: (0,) if linear; (0,1,2) if parallel] Performs the specified stages of the start of a `Piper` instance. Stage 0 is necessary and sufficient to start a linear `Piper` which uses an `itertools.imap`. Stages 1 and 2 are required to start any parallel `Piper` instance.

stop (*forced=False, **kwargs*)

Attempts to cleanly stop the `Piper` instance. A `Piper` is “started” if its `NuMap` instance is “started”. Non-parallel `Pipers` do not have to be started or stopped. An `NuMap` instance can be stopped by triggering its stopping procedure and retrieving results from the `NuMaps` end tasks. Because neither the `Piper` nor the `NuMap` “knows” which tasks i.e. `Pipers` are the end tasks they have to be specified:

```
end_task_ids = [0, 1]    # A list of NuMap task ids
piper_instance.stop(ends =end_task_ids)
```

results in:

```
NuMap_instance.stop(ends =[0,1])
```

If the `Piper` did not finish processing the data before the `stop` method is called the “forced” argument has to be `True`:

```
piper_instance.stop(forced =True, ends =end_task_ids)
```

If the `Piper` (and consequently `NuMap`) is part of a `Dagger` graph the `Dagger.stop` method should be called instead. See: `NuMap.stop` and `Dagger.stop`.

verify this: # If “forced” is set `True` but the `ends NuMap` argument is not # given. The `NuMap` instance will not try to retrieve any results and # will not call the `NuMap._stop` method.

Arguments:

- `forced(bool)` [default: `False`] The Piper will be forced to stop the NuMap instance.

Additional keyworded arguments are passed to the `Piper.imap` instance.

exception `papy.core.PiperError`

Exceptions raised or related to Piper instances.

class `papy.core.Plumber` (*logger_options={}*, ***kwargs*)

The Plumber is a subclass of Dagger and Graph with added run-time methods and a high-level interface for working with PaPy pipelines.

Arguments:

- `dagger(Dagger instance)` [default: `None`] An optional Dagger instance. if `None` a new one is created.

load (*filename*)

Instantiates (loads) pipeline from a source code file.

Arguments:

- `filename(path)` location of the pipeline source code.

pause ()

Pauses a running pipeline. This will stop retrieving results from the pipeline. Parallel parts of the pipeline will stop after the NuMap buffer is has been filled. A paused pipeline can be run or stopped.

run ()

Executes a started pipeline by pulling results from it's output Pipers. Processing nodes i.e. Pipers with the `track` attribute set `True` will have their returned results stored within the `Dagger.stats['pipers_tracked']` dictionary. A running pipeline can be paused.

save (*filename*)

Saves pipeline as a Python source code file.

Arguments:

- `filename(path)` Path to save the pipeline source code.

start (*datas*)

Starts the pipeline by connecting the input Pipers of the pipeline to the input data, connecting the pipeline and starting the NuMap instances.

The order of items in the “datas” argument sequence should correspond to the order of the input Pipers defined by `Dagger._cmp` and `Piper.ornament`.

Arguments:

- `datas(sequence)` A sequence of external input data in the form of sequences or iterators.

stop ()

Stops a paused pipeline. This will a trigger a `StopIteration` in the inputs of the pipeline. And retrieve the buffered results. This will stop all Pipers and NuMaps. Python will not terminate cleanly if a pipeline is running or paused.

wait (*timeout=None*)

Waits (blocks) until a running pipeline finishes.

Arguments:

- `timeout(int)` [default: `None`] Specifies the timeout, `RuntimeError` will be raised. The default is to wait indefinitely for the pipeline to finish.

exception `papy.core.PlumberError`

Exceptions raised or related to `Plumber` instances.

class `papy.core.Worker` (*functions, arguments=None, kwargs=None, name=None*)

The `Worker` is an object that composes sequences of functions. When called these functions are evaluated from left to right. The function on the right will receive the return value from the function on the left.

The constructor takes optionally sequences of positional and keyworded arguments for none or all of the composed functions. Positional arguments should be given in a tuple. Each element of this tuple should be a tuple of positional arguments for the corresponding function. If a function does not take positional arguments its corresponding element in the arguments tuple should be an empty tuple i.e. (). Keyworded arguments should also be given in a tuple. Each element of this tuple should be a dictionary of arguments for the corresponding function. If a function does not take any keyworded arguments its corresponding element in the keyworded arguments tuple should be an empty dict i.e. {}. If none of the functions takes arguments of a given type the positional and/or keyworded arguments tuple can be omitted.

All exceptions raised by the functions are caught, wrapped and returned **not** raised. If the `Worker` is called with the first argument being a sequence which contains an `Exception` no function is evaluated and the `Exception` is re-wrapped and returned.

A `Worker` instance can be constructed in a variety of ways:

- with a sequence of functions and a optional sequences of positional and keyworded arguments e.g.:

```
Worker((func1,          func2,    func3),
       ((arg11, arg21), (arg21,)), ()),
       ({}),          {},          {'arg31':arg31}))
```

- with another `Worker` instance, which results in their functional equivalence e.g.:

```
Worker(worker_instance)
```

- with multiple `Worker` instances, where the functions and arguments of the `Workers` are combined e.g.:

```
Worker((worker1, worker2))
```

this is equivalent to:

```
Worker(worker1.task + worker2.task,          worker1.args + worker2.args,
```

- with a single function and its arguments in a tuple e.g.:

```
Worker(function, (arg1, arg2, arg3))
```

this is equivalent to:

```
Worker((function,), ((arg1, arg2, arg3),))
```

exception `papy.core.WorkerError`

Exceptions raised or related to `Worker` instances.

1.16.2 `papy.graph`

This module implements a graph data structure without explicit edges, using nested Python dictionaries. It provides `DictNode` and `DictGraph`.

class `papy.graph.DictGraph` (*nodes=(), edges=(), xtras=None*)

A dictionary-based graph data structure. This graph implementation is a little bit unusual as it does not explicitly hold a list of edges. A `DictGraph` instance is a dictionary, where the keys of the dictionary are hashable object instances (**node objects**), while the values are `DictNode` instances (**topological nodes**).

A `DictNode` instance is also a dictionary, where the keys are **node objects** and the values are `DictNode` instances. A `Node` instance (value) is basically a dictionary of outgoing edges from the **node object** (key). The edges are indexed by the incoming objects. So we end up with a single recursively nested dictionary which defines the topology of the `DictGraph` instance. An edge is a tuple of two **node objects**.

Arguments:

- `nodes(iterable)` [default: `()`] A sequence of **node objects** to be added to the graph. See: `Graph.add_nodes`
- `edges(iterable)` [default: `()`] A sequence of edges to be added to the graph. See: `Graph.add_edges`
- `xtras(iterable)` [default: `None`] A sequence of property dictionaries for the added **node objects**. The **topological nodes** corresponding to the added **node objects** will have their `Node.xtra` attributes updated with the contents of this sequence. Either all or no "xtra" dictionaries must be given.

`add_edge` (*edge*, *double=False*)

Adds an edge to the `DictGraph`. An edge is just a pair of **node objects**. If the **node objects** are not in the graph they are created.

Arguments:

- `edge(iterable)` An ordered pair of **node objects**. The edge is assumed to have a direction from the first to the second **node object**.
- `double(bool)` [default: `False`] If `True` the the reverse edge is also added.

`add_edges` (*edges*, **args*, ***kwargs*)

Adds edges to the graph. Takes optional arguments for `DictGraph.add_edge`.

Arguments:

- `edges(iterable)` Sequence of edges to be added to the `DictGraph`.

`add_node` (*node*, *xtra=None*, *branch=None*)

Adds a **node object** to the `DictGraph`. Returns `True` if a new **node object** has been added. If the **node object** is already in the `DictGraph` returns `False`.

Arguments:

- `node(object)` Node to be added. Any hashable Python object.
- `xtra(dict)` [default: `None`] The newly created topological `Node.xtra` dictionary will be updated with the contents of this dictionary.
- `branch(object)` [default: `None`] an identifier used to sort topologically equivalent branches.

`add_nodes` (*nodes*, *xtras=None*)

Adds **node objects** to the graph.

Arguments:

- `nodes(iterable)` Sequence of **node objects** to be added to the `DictGraph`
- `xtras(iterable)` [default: `None`] Sequence of `Node.xtra` dictionaries corresponding to the **node objects** being added. See: `Graph.add_node`.

`clear_nodes` ()

Clears all nodes in the `Graph`. See `Node.clear`.

`cmp_branch` (*node1*, *node2*)

comparison of **node objects** based on the "branch" attribute of their **topological nodes**.

`deep_nodes` (*node*)

Returns all reachable **node objects** from a **node object**. See: `DictNode.deep_nodes`.

Arguments:

- `node(object)` a **node object** present in the graph.

del_edge (*edge*, *double=False*)

Removes an edge from the `DictGraph`. An edge is a pair of **node objects**. The **node objects** are not removed from the `DictGraph`.

Arguments:

- `edge(tuple)` An ordered pair of **node objects**. The edge is assumed to have a direction from the first to the second **node object**.
- `double(bool)` [default: `False`] If `True` the the reverse edge is also removed.

del_edges (*edges*, **args*, ***kwargs*)

Removes edges from the graph. Takes optional arguments for `DictGraph.del_edge`.

Arguments:

- `edges(iterable)` Sequence of edges to be removed from the `DictGraph`.

del_node (*node*)

Removes a **node object** from the `DictGraph`. Returns `True` if a **node object** has been removed. If the **node object** is not in the `DictGraph` raises a `KeyError`.

Arguments:

- `node(object)` **node object** to be removed. Any hashable Python object.

del_nodes (*nodes*)

Removes **node objects** from the graph.

Arguments:

- `nodes(iterable)` Sequence of **node objects** to be removed from the `DictGraph`. See: `DictGraph.del_node`.

dfs (*node*, *bucket=None*, *order='append'*)

Recursive depth first search. By default (`"order" = "append"`) this returns the **node objects** in the reverse postorder. To change this into the preorder use a `collections.deque` as `"bucket"` and `"appendleft"` as `"order"`.

Arguments:

- `bucket(list or collections.dequeue)` [default: `None`] The user *must* provide a new list or `collections.dequeue` to store the nodes.
- `order(str)` [default: `"append"`] Method of the `"bucket"` which will be called with the **node object** that has been examined. Other valid options might be `"appendleft"` for a `collections.dequeue`.

edges (*nodes=None*)

Returns a tuple of all edges in the `DictGraph` an edge is a pair of **node objects**.

Arguments:

- `nodes(iterable)` [default: `None`] iterable of **node objects** if specified the edges will be limited to those outgoing from one of the specified nodes.

incoming_edges (*node*)

Returns a tuple of incoming edges for a **node object**.

Arguments:

- `node(object)` **node object** present in the graph to be queried for incoming edges.

iter_nodes ()

Returns an iterator of all **node objects** in the DictGraph.

node_rank ()

Returns the maximum rank for each **topological node** in the DictGraph. The rank of a node is defined as the number of edges between the node and a node which has rank 0. A **topological node** has rank 0 if it has no incoming edges.

node_width ()

Returns the width of each node in the graph. #TODO

nodes ()

Returns a list of all **node objects** in the DictGraph.

outgoing_edges (*node*)

Returns a tuple of outgoing edges for a **node object**.

Arguments:

- node(object) **node object** present in the graph to be queried for outgoing edges.

postorder ()

Returns a valid postorder of the **node objects** of the DictGraph *if* the topology is a directed acyclic graph. This postorder is semi-random, because the order of elements in a dictionary is semi-random and so are the starting nodes of the depth-first search traversal, which determines the postorder, consequently some postorders will be discovered more frequently.

This postorder enforces some determinism on particular ties:

- topologically equivalent branches come first are sorted by length (shorter branches come first).
- if the topological Nodes corresponding to the node objects have a "branch" attribute it will be used to sort the graph from left to right.

However the final postorder is still *not* deterministic.

rank_width ()

Returns the width of each rank in the graph. #TODO

class papy.graph.**DictNode** (*entity=None, xtra=None*)

DictNode is the **topological node** of a DictGraph. Please note that the **node object** is not the same as the **topological node**. The **node object** is any hashable Python object. The **topological node** is defined for each **node object** and is a dictionary of other **node objects** with incoming **edges** from a single **node object**.

A node has: "discovered", "examined" and "branch" attributes.

Arguments:

- entity (object) [default: None] Any hashable object is a valid **node object**.
- xtra (dict) [default: None] A dictionary of arbitrary properties of the **topological node**.

clear ()

Sets the "discovered" and "examined" attributes to False.

deep_nodes (*allnodes=None*)

A recursive method to return a list of *all* **node objects** connected from this **topological node**.

iternodes ()

Returns an iterator of **node objects** directly connected from this **topological node**.

nodes ()

Returns a list of **node objects** directly connected from this **topological node**.

1.16.3 `papy.util.codefile`

Provides template strings for saving **PaPy** pipelines directly as Python source code.

1.16.4 `papy.util.config`

Configures logging to monitor the execution of **PaPy** pipelines and OS-dependent defaults for different variables.

`papy.util.config.get_defaults()`

Returns a dictionary of variables and their possibly os-dependent defaults.

`papy.util.config.start_logger(log_to_file=False, log_to_stream=False, log_to_file_level=20, log_to_stream_level=20, log_filename=None, log_stream=None, log_rotate=True, log_size=524288, log_number=3)`

Configures and starts a logger to monitor the execution of a **PaPy** pipeline.

Arguments:

- `log_to_file(bool)` [default: `True`] Should we write logging messages into a file?
- `log_to_stream(bool or object)` [default: `False`] Should we print logging messages to a stream? If `True` this defaults to `stderr`.
- `log_to_file_level(int)` [default: `INFO`] The minimum logging level of messages to be written to file.
- `log_to_stream_level(int)` [default: `ERROR`] The minimum logging level of messages to be printed to the stream.
- `log_filename(str)` [default: `"PaPy_log"` or `"PaPy_log_${TIME$}"`] Name of the log file. Ignored if `"log_to_file"` is `False`.
- `log_rotate(bool)` [default: `True`] Should we limit the number of logs? Ignored if `"log_to_file"` is `False`.
- `log_size(int)` [default: `524288`] Maximum number of bytes saved in a single log file. Ignored if `"log_to_file"` is `False`.
- `log_number(int)` [default: `3`] Maximum number of rotated log files Ignored if `"log_to_file"` is `False`.

1.16.5 `papy.util.runtime`

Provides a (possibly shared, but not yet) dictionary.

`papy.util.runtime.get_runtime()`

Returns a `PAPY_RUNTIME` dictionary.

INDICES AND TABLES

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

PYTHON MODULE INDEX

p

`papy.core`, 36
`papy.graph`, 42
`papy.util.codefile`, 45
`papy.util.config`, 46
`papy.util.runtime`, 46

INDEX

A

add_edge() (papy.graph.DictGraph method), 43
add_edges() (papy.graph.DictGraph method), 43
add_node() (papy.graph.DictGraph method), 43
add_nodes() (papy.graph.DictGraph method), 43
add_pipe() (papy.core.Dagger method), 36
add_piper() (papy.core.Dagger method), 36
add_pipers() (papy.core.Dagger method), 36
add_pipes() (papy.core.Dagger method), 36

C

children_after_parents() (papy.core.Dagger method), 37
clear() (papy.graph.DictNode method), 45
clear_nodes() (papy.graph.DictGraph method), 43
cmp_branch() (papy.graph.DictGraph method), 43
connect() (papy.core.Dagger method), 37
connect() (papy.core.Piper method), 39
connect_inputs() (papy.core.Dagger method), 37

D

Dagger (class in papy.core), 36
DaggerError, 38
deep_nodes() (papy.graph.DictGraph method), 43
deep_nodes() (papy.graph.DictNode method), 45
del_edge() (papy.graph.DictGraph method), 44
del_edges() (papy.graph.DictGraph method), 44
del_node() (papy.graph.DictGraph method), 44
del_nodes() (papy.graph.DictGraph method), 44
del_pipe() (papy.core.Dagger method), 37
del_piper() (papy.core.Dagger method), 37
del_pipers() (papy.core.Dagger method), 37
del_pipes() (papy.core.Dagger method), 38
dfs() (papy.graph.DictGraph method), 44
DictGraph (class in papy.graph), 42
DictNode (class in papy.graph), 45
disconnect() (papy.core.Dagger method), 38
disconnect() (papy.core.Piper method), 40

E

edges() (papy.graph.DictGraph method), 44

G

get_defaults() (in module papy.util.config), 46
get_inputs() (papy.core.Dagger method), 38
get_outputs() (papy.core.Dagger method), 38
get_runtime() (in module papy.util.runtime), 46

I

incoming_edges() (papy.graph.DictGraph method), 44
iter_nodes() (papy.graph.DictGraph method), 44
iternodes() (papy.graph.DictNode method), 45

L

load() (papy.core.Plumber method), 41

N

next() (papy.core.Piper method), 40
node_rank() (papy.graph.DictGraph method), 45
node_width() (papy.graph.DictGraph method), 45
nodes() (papy.graph.DictGraph method), 45
nodes() (papy.graph.DictNode method), 45

O

outgoing_edges() (papy.graph.DictGraph method), 45

P

papy.core (module), 36
papy.graph (module), 42
papy.util.codefile (module), 45
papy.util.config (module), 46
papy.util.runtime (module), 46
pause() (papy.core.Plumber method), 41
Piper (class in papy.core), 38
PiperError, 41
Plumber (class in papy.core), 41
PlumberError, 41
postorder() (papy.graph.DictGraph method), 45

R

rank_width() (papy.graph.DictGraph method), 45
resolve() (papy.core.Dagger method), 38

run() (papy.core.Plumber method), 41

S

save() (papy.core.Plumber method), 41

start() (papy.core.Dagger method), 38

start() (papy.core.Piper method), 40

start() (papy.core.Plumber method), 41

start_logger() (in module papy.util.config), 46

stop() (papy.core.Dagger method), 38

stop() (papy.core.Piper method), 40

stop() (papy.core.Plumber method), 41

W

wait() (papy.core.Plumber method), 41

Worker (class in papy.core), 42

WorkerError, 42

NuMap Documentation

Release 1.0

July 30, 2010

CONTENTS

1	Examples	3
2	NuMap API	5
2.1	numap.NuMap	5
3	Indices and tables	11
	Python Module Index	13
	Index	15

NuMap is a parallel (thread- or process-based, local or remote), buffered, multi-task, `itertools.imap` or `multiprocessing.Pool.imap` function replacment. Like `imap` it evaluates a function on elements of a sequence or iterable, and it does so lazily. Laziness can be adjusted via the “stride” and “buffer” arguments. Unlike `imap`, NuMap supports **multiple pairs** of function and iterable **tasks**. The **tasks** are **not** queued rather they are **interwoven** and share a pool or **worker** “processes” or “threads” and a memory “buffer”.

The package is tested on Python 2.6+

Contents:

EXAMPLES

Examples can be found in the `doc/examples` directory of the source distribution. See also `test/test_numap.py` to explore the API.

NUMAP API

NuMap is a parallel (local or remote), buffered, multi-task, lazy map function, which can use threads and processes.

2.1 numap.NuMap

This module provides a parallel (local or remote), buffered, multi-task, lazy map function, which can use threads and processes.

class numap.NuMap.**NuMap** (*func=None, iterable=None, args=None, kwargs=None, worker_type=None, worker_num=None, worker_remote=None, stride=None, buffer=None, ordered=True, skip=False, name=None*)

NuMap is a parallel (thread- or process-based, local or remote), buffered, multi-task, `itertools.imap` or `multiprocessing.Pool.imap` function replacement. Like `imap` it evaluates a function on elements of a sequence or iterable, and it does so lazily. Laziness can be adjusted via the “stride” and “buffer” arguments. Unlike `imap`, NuMap supports **multiple pairs** of function and iterable **tasks**. The **tasks** are **not** queued rather they are **interwoven** and share a pool or **worker** “processes” or “threads” and a memory “buffer”.

Pool

The pool is a set of managed **worker** processes or threads. The choice of the “worker_type” has a **fundamental** impact on the performance of the map. As a general rule use “process” if you have multiple CPUs or CPU-cores and your task functions are cpu-bound. Use “thread” if your function is IO-bound e.g. retrieves data from the Web. Increasing the number of **workers** above the number of CPUs makes sense only if these are “thread” based **workers** and the evaluated functions are IO-bound. Some CPU-bound tasks might evaluate faster if the number of **worker** processes equals the number of CPUs + 1. For “thread” based NuMaps a larger number of **workers** might improve performance. The “worker_num” argument must not **not** include workers needed to run remote processes and can be equal 0 for a purely remote NuMaps.

Iteration

Results are retrieved through iteration. A single NuMap instance supports iteration over results from many **tasks**. This means that it supports multiple end-points. The default is to iterate over the results from the first task. An iterator for a single **task** is returned by the `NuMap.get_task` method.

Order

The tasks can be interdependent i.e. the results from one **task** being the input to a second **task**. The order in which **tasks** are added to the NuMap instance is important. It affects the order of evaluation and consequently the order in which results should be retrieved. If the **tasks** are chained then the “order” must be a valid topological sort (reverse topological order). If the NuMap is ordered the n-th result for a specific **task** will be always be available before the n+1-th result. If “order” is `False` the results will be available in the order they are calculated.

Skipping

The “skipping” argument determines how to respond to `TimeoutErrors` it is ignored if no “timeout” value is given to the `NuMap.next` method. If “skipping” is `True` results, which are not calculated on time will be omitted. If “skip” `False` an exception will be raised, but the result can be retrieved later. If **tasks** are chained a `TimeoutError` will collapse the `NuMap` evaluation. Do **not* specify timeouts in for chained **tasks**.

Parallel evaluation

The parallelism of the evaluation is strictly defined by the “stride”, “buffer” and the total number of **workers** in the pool. The **worker** number is obviously the upper bound of concurrently evaluated elements. The maximum number of elements from a single **task** evaluated in parallel is defined by “stride”. The “buffer” limits the maximum number of pending results for all **tasks** it is a function of “stride”, but also of the topology of dependencies between the tasks. A long “stride” improves parallelism, but increases “buffer” memory requirements. It should not be smaller than the number of pool **workers**, because some will be idle. The size of the “buffer” is larger or equal to “stride” because a **task** might depend on results from multiple up-stream **tasks**.

The minimum “stride” and “buffer is 1 therefore the results from the “buffer” must be removed in the same order as input elements are evaluated. Otherwise the `NuMap` might dead-lock.

An element is buffered until it is returned by the `NuMap.next` method. Starting the `NuMap` will cause one element (the first from the first **task** to be submitted to the pool. For “stride” equal to 1, the next queued element is the first from the second **task**, which can enter the pool only if either the first result is retrieved (i.e. `NuMap.next` returns) or the “buffer” is larger then the “stride”. If the “buffer” is `n` then `n` tasklets can enter the pool. A “stride” of `n` requires at least `n` elements to enter the pool, therefore “buffer” cannot be smaller then “stride”. The “minimum” buffer is the maximum possible number of queued results. This number depends on the interdependencies between **tasks** and the “stride”. The default is conservative and sufficient for all topologies. If the **tasks** are chained i.e. the output from one is consumed by another then at most one `i`-th element from each chained **task** is at a given moment in the pool. In those cases the minimum “buffer” to satisfy the worst case number of queued results is lower then the safe default.

Stopping

The `NuMap` can be stopped at any time, however some buffered results might be lost and up to `2 * “stride”` additional input elements consumed. If pending buffered results are not retrieved the `NuMap` might not shut down properly.

Arguments:

- `func` (callable) [default: `None`] If the `NuMap` is given a function it is used to define the first and only task of the `NuMap`
- `iterable` (iterable) [default: `None`] a sequence of first arguments for “func” required if “func” is given
- `args` (tuple) [default: `None`] optional, see: `NuMap.add_task`
- `kwargs` (dict) [default: `None`] optional, see: `NuMap.add_task`
- `worker_type` ('process' or 'thread') [default: 'process'] Defines the type of internally spawned pool workers. For `multiprocessing.Process` based worker choose 'process' for threading. `Thread` workers choose 'thread'.
- `worker_num`(int) [default: number of CPUs, min: 1] The number of workers to spawn locally. Defaults to the number of available CPUs, which is a reasonable choice for process-based `NuMaps`.
- `worker_remote`(iterable) [default: `None`] A sequence of “remote host” “remote worker_num” tuples e.g. `((‘localhost’, 2]), (‘127.0.0.1’, 2))` “remote worker_num” is the number of workers processes per remote host. A custom TCP port can be specified `((‘localhost:6666’, 2),)`.
- `stride`(int) [default: automatic] number of elements from a **task** evaluated in parallel

- `buffer(int)` [default: automatic] total number number of elements (inputs and results) in the NuMap instance
- `ordered(bool)` [default: True] If True the output of all **tasks** will be ordered see: `order`.
- `skip(bool)` [default: False] Should we skip a result if trying to retrieve it raised a `TimeoutError`?
- `name(str)` [default: "imap_id(id(object))"] an optional name to associate with this NuMap instance. It should be unique. Useful for code generation.

Restrictions:

- a completely lazy i.e. "buffer-free" evaluation is not supported
- if remote workers are enabled, "worker_type" has to be the default "process".

add_task (*func, iterable, args=None, kwargs=None, timeout=None, block=True, track=False*)

Adds a **task** to evaluate. A **task** is jointly a function or callable an iterable with optional arguments and keyworded arguments. The iterable can be the result iterator of a previously added **task** (to the same or to a different NuMap instance).

Arguments:

- `func` (callable) this will be called on each element of the "iterable" and supplied with arguments "args" and keyworded arguments "kwargs"
- `iterable` (iterable) this must be a sequence of *picklable* objects which will be the first arguments passed to the "func"
- `args` (tuple) [default: None] A tuple of optional constant arguments passed to the callable "func" after the first argument from the "iterable"
- `kwargs` (dict) [default: None] A dictionary of keyworded arguments passed to "func" after the variable argument from the "iterable" and the arguments from "args"
- `timeout` (bool) see: `_NuMapTask`
- `block` (bool) see: `_NuMapTask`
- `track` (bool) [default: False] If True the results (or exceptions) of a **task** are saved within: `self._tasks_tracked[%task_id%]` as a `{index:result}` dictionary. This is only useful if the callable "func" creates persistent data. The dictionary can be used to restore the correct order of the data

get_task (*task=0, timeout=None, block=True*)

Returns an iterator which results are limited to one **task**. The default iterator the one which e.g. will be used in a for loop is the iterator for the first task (`task=0`). The returned iterator is a `_NuMapTask` instance.

Compare:

```
for result_from_task_0 in imap_instance:
    pass
```

with:

```
for result_from_task_1 in imap_instance.get_task(task_id =1):
    pass
```

a typical use case is:

```
task_0_iterator = imap_instance.get_task(task_id =0)
task_1_iterator = imap_instance.get_task(task_id =1)
```

```
    for (task_1_res, task_0_res) in izip(task_0_iterator, task_1_iterator):  
        pass
```

next (*timeout=None, task=0, block=True*)

Returns the next result for the given **task**. Defaults to 0, which is the first **task**. If multiple chained tasks are evaluated then the next method of only the last should be called directly.

Arguments:

- **timeout** (*float*) Number of seconds to wait until a `TimeoutError` is raised.
- **task** (*int*) id of the task from the `NuMap` instance
- **block** (*bool*) if `True` call will block until result is available

pop_task (*number*)

Removes a previously added **task** from the `NuMap` instance.

Arguments:

- **number** (*int* or `True`) A positive integer specifying the number of **tasks** to pop. If number is set `True` all **tasks** will be popped.

start (*stages=(1, 2)*)

Starts the processes or threads in the internal pool and the threads, which manage the **worker pool** input and output queues. The **starting mode** is split into **two stages**, which can be initiated separately. After the first stage the **worker pool** processes or threads are started and the `NuMap._started` event is set `True`. A call to the `NuMap.next` method **will** block. After the **second stage** the `NuMap._pool_putter` and `NuMap._pool_getter` threads will be running. The `NuMap.next` method should only be called **after** this method returns.

Arguments:

- **stages** (*tuple*) [default: (1, 2)] Specifies which stages of the start process to execute, by default both stages.

stop (*ends=None, forced=False*)

Stops an `NuMap` instance. If the list of end tasks is specified *via* the “ends” argument a call to `NuMap.stop` will block the calling thread and retrieve (discards) a maximum of $2 * \text{stride}$ of results. This will stop the worker pool and the threads which manage its input and output queues respectively.

If the “ends” argument is not specified, but the “forced” argument is the method does not block and the `NuMap._stop` has to be called after **all** pending results have been retrieved. Calling `NuMap._stop` with pending results **will** dead-lock.

Either “ends” or “forced” has to be `True`.

Arguments:

- **ends** (*list*) [default: `None`] A list of task ids which are not consumed within the `NuMap` instance.
- **forced** (*bool*) [default: `False`] If “ends” is not `None` this argument is ignored. If “ends” is `None` and “forced” is `True` the `NuMap` instance will trigger *stopping mode*.

`numap.NuMap`. **imports** (*modules, forgive=False*)

Should be used as a decorator to *attach* import statements to function definitions. These imports are added to the global (i.e. module-level of the decorated function) namespace.

Two forms of import statements are supported (in the following examples `foo`, `bar`, `oof`, and ```rab` are modules not classes or functions):

```
import foo, bar           # -> @imports(['foo', 'bar'])
import foo.oof as oof    # -> @imports(['foo.oof', 'bar.rab'])
import bar.rab as rab
```

It provides support for alternatives:

```
try:
    import foo
except ImportError:
    import bar
```

which is expressed as:

```
@imports(['foo,bar'])
```

or alternatively:

```
try:
    import foo.oof as oof
except ImportError:
    import bar.rab as oof
```

becomes:

```
@imports(['foo.oof,bar.rab'])
```

This import is available in the body of the function as `oof`. All needed imports should be attached for every function (even if two function are in the same module and have the same globals)

Arguments:

- `modules` (list) A list of modules in the following forms `['foo', 'bar', ..., 'baz']` or `['foo.oof', 'bar.rab', ..., 'baz.zab']`
- `forgive` (bool) [default: False] If True will not raise *ImportError*

INDICES AND TABLES

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

PYTHON MODULE INDEX

n

`numap.NuMap`, 5

INDEX

A

`add_task()` (`numap.NuMap.NuMap` method), 7

G

`get_task()` (`numap.NuMap.NuMap` method), 7

I

`imports()` (in module `numap.NuMap`), 8

N

`next()` (`numap.NuMap.NuMap` method), 8

`NuMap` (class in `numap.NuMap`), 5

`numap.NuMap` (module), 5

P

`pop_task()` (`numap.NuMap.NuMap` method), 8

S

`start()` (`numap.NuMap.NuMap` method), 8

`stop()` (`numap.NuMap.NuMap` method), 8

NuBio Documentation

Release 1.0

July 30, 2010

CONTENTS

1	Concepts	3
2	Contents	5
2.1	Data Containers	5
2.2	Data Formats	5
2.3	Biological Data	5
2.4	Utility Modules	5
2.5	Examples	6
2.6	NuBio API	6
3	Indices and tables	15
	Python Module Index	17
	Index	19

The `nubio` package provides data-structures to store and manipulate biological entities i.e. sequences, alignments and molecular structures; useful biological data e.g. genetic codes and functions to read and write common file formats e.g. “pdb” or “newick”. It is based on the `NuBox` hierarchical array from the `nubox` package.

The data containers are based on a hierarchical array concept. Raw data is stored in a flat array, but its aspect depends on a facade object. For example “ATGGCG” can act as a `NtSeq` (sequence of 6 nucleotides) or `CodonSeq` (sequence of 2) codons. This allows to customize the behaviour of objects and the storage of metadata at multiple hierarchical levels.

The package is tested on Python 2.6+, Jython 2.5+

CONCEPTS

NuBio containers are specific sub-classes of typed `NuBox` classes. For example `NtSeq` is a specific sub-class of `CharVector` with methods valid for nucleotide data e.g. `complement`.

Please see the documentation for the **NuBox** for additional details.

CONTENTS

2.1 Data Containers

NuBio includes containers for common bioinformatics datatypes these currently include:

- `Aa`, `AaSeq` and `AaAln` for holding amino acid symbols, sequences and alignments. These are based on generics from `nubox.char`.
- `Nt`, `NtSeq` and `NtAln` for holding nucleotide symbols, sequences and alignments. These are based on generics from `nubox.char`.
- `Codon`, `CodonSeq` and `CodonAln` for holding codons, sequences and alignments of codons. These are based on generics from `nubox.char`.
- `Atom` and `Model` for holding coordinates of atoms and larger molecular structures. These are based on generics from `nubox.double`.
- `AtomData` and `ModelData` for holding data from PDB fields of atoms and larger molecular structures.

2.2 Data Formats

General and utility functions to parse or write bioinformatics file formats are provided in: `nubio.io`.

Currently supported file-formats:

- `fasta` (sequences and alignments) available in `nubio.io.fasta`
- `stockholm` (annotated alignments) available in `nubio.io.stockholm`
- `PDB` (macromolecular structures) available in `nubio.io.pdb`

2.3 Biological Data

Currently Included are biological alphabets (IUPAC), genetic codes (NCBI) and score matrices (BLOSUM, PAM). All are available from `nubio.data`, functions to parse the raw files are also available.

2.4 Utility Modules

Currently included utility modules are providing functionality for the interaction with:

- web pages and services `nubio.util.url`
- command-line applications `nubio.util.app`

2.5 Examples

Examples can be found in the `doc/examples` directory of the source distribution. See also `test/test_*.py` to explore the API.

2.6 NuBio API

2.6.1 `nubio.core`

The `nubio.core` module provides bioinformatics data structures.

2.6.2 `nubio.core.aa`

The `nubio.core.aa` module provides classes to represent amino acids, protein sequences and their alignments.

class `nubio.core.aa.Aa` (**args*, ***kwargs*)

A subclass of `Char` to represent an amino acid.

The class metadata can contain an `ALPHABET` that defines the valid symbols by default `AA_IUPAC`. See: `nubio.data.alphabet`.

clsparent

alias of `AaSeq`

class `nubio.core.aa.AaAln` (**args*, ***kwargs*)

Represents an alignment of amino acids.

clschild

alias of `AaSeq`

class `nubio.core.aa.AaSeq` (**args*, ***kwargs*)

Represents a vector of amino acids.

clschild

alias of `Aa`

clsparent

alias of `AaAln`

2.6.3 `nubio.core.nt`

Provides classes to represent nucleotides, codons, nucleic acid sequences and alignments.

class `nubio.core.nt.Codon` (**args*, ***kwargs*)

A subclass of `Nt` to represent a codon.

The class metadata can contain an `ALPHABET`, `COMPLEMENT_CODE` and `GENETIC_CODE` that define the valid symbols and their complements respectively. The defaults are `NT_COMPLEMENT_IUPAC`, `NT_IUPAC` and `GENETIC_CODE_STANDARD`. see: `nubio.data.alphabet`.

clsparentalias of `CodonSeq`**translate** (*code=None, strict=False, **kwargs*)Returns the translation given a genetic “code” in a `AaSeq`.

Arguments:

- `code(dict)` [default: `None`] the default complement code is `GENETIC_CODE_STANDARD`.
- `strict(bool)` [default: `False`] if `True` will require a valid start codon and terminate on stop codons.

class `nubio.core.nt.CodonAln` (**args, **kwargs*)

Represents an alignment of codons.

clschildalias of `CodonSeq`**class** `nubio.core.nt.CodonSeq` (**args, **kwargs*)

Represents a vector of codons.

clschildalias of `Codon`**clsparent**alias of `CodonAln`**class** `nubio.core.nt.Nt` (**args, **kwargs*)A subclass of `Char` to represent a nucleotide.

The class metadata can contain an `ALPHABET` and `COMPLEMENT_CODE` that define the valid symbols and their complements respectively. The defaults are `NT_COMPLEMENT_IUPAC` and `NT_IUPAC` see: `nubio.data.alphabet`.

clsparentalias of `NtSeq`**complement** (*code=None, inplace=False*)

Returns the complement given a complement “code”. Optionally modifies the object inplace.

Arguments:

- `code(dict)` [default: `None`] the default complement code is `NT_COMPLEMENT_IUPAC`.
- `inplace(bool)` [default: `False`] if `True` the array will be modified in-place.

class `nubio.core.nt.NtAln` (**args, **kwargs*)

Represents a vector of nucleotides.

clschildalias of `NtSeq`**class** `nubio.core.nt.NtSeq` (**args, **kwargs*)

Represents a vector of nucleotides.

clschildalias of `Nt`**clsparent**alias of `NtAln`

2.6.4 nubio.core.atom

The `nubio.core.atom` module provides support for 3D structures: atoms, molecules and macromolecules. It includes array containers for coordinate data and PDB fields.

class `nubio.core.atom.Atom` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
Represents a point in 3d Cartesian coordinates.

class `nubio.core.atom.AtomData` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
A `NuBoxScalar` with a structured data type to store the non-coordinate fields in a “ATOM” or “HETATM” PDB-line. See: `nubio.io.pdb` for details. fields:

- ('at_type', 'S6'),
- ('ser_num', 'int32'),
- ('at_id', 'S4'),
- ('alt_loc', 'S1'),
- ('res_name', 'S3'),
- ('chain_id', 'S1'),
- ('res_id', 'int32'),
- ('res_ic', 'S1'),
- ('occupancy', 'float64'),
- ('bfactor', 'float64'),
- ('element', 'S2'),
- ('charge', 'S2')

class `nubio.core.atom.Model` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
Represents a vector of `Atom` instances.

clschild
alias of `Atom`

class `nubio.core.atom.ModelData` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
Represents a vector of `AtomData`. See: `AtomData`.

clschild
alias of `AtomData`

2.6.5 nubio.data

The `nubio.data` module provides biological data as simple Python types. Alphabets, genetic codes and score matrices are `tuple`, `dict` and nested `dict` instances.

2.6.6 nubio.data.alphabet

The `nubio.data.alphabet` module provides biological alphabets as tuples of NuBox items. See: `nubox.char.Char`.

2.6.7 nubio.data.score

The `nubio.data.score` module provides pairwise scoring matrices.

`nubio.data.score.exponent_score_matrix` (*sm*, *scale=1.0*)

Exponentiates a score matrix object “sm” and optionally scales it by “scale”.

Arguments:

- `sm(dict)` a scoring matrix dictionary see: `parse_score_matrix`.
- `scale(float)` [default: 1.0] an optional scale factor.

`nubio.data.score.parse_blosum_out` (*fh*)

Parser for a blosumXX.out file. Returns a dictionary.

Arguments:

- `fh(file)` A file-like object open for reading.

`nubio.data.score.parse_score_matrix` (*fh*)

Parses a score matrix, which in the format of the matrices available at: <ftp://ftp.ncbi.nih.gov/blast/matrices/>. Code from adapted from: <http://bitbucket.org/brentp/biostuff/src/tip/nwalign/pairwise.py> Author: brentp <bpederse at gmail com>. Returns a nested `dict` representation of the 2D-matrix.

Arguments:

- `fh(file)` A file-like object open for reading.

2.6.8 nubio.data.files

The `nubio.data.files` contains raw files of biological data embedded in Python source code. Currently the most common score matrices are included.

2.6.9 nubio.io

The `nubio.io` module provides parsers and writers for common biological data file formats.

`nubio.io.aln_parser` (*fh*, *vector*, ***kwargs*)

Returns an alignment of sequences in a file-like object as an instance of the given “vector”. No grand-children corresponding to symbols in the alignment are constructed.

Arguments:

- `fh(file)` A file-like object open for reading.
- `vector(nubox.char.CharArray)` a subclass of `nubox.char.CharArray`

Additional keyworded arguments are updated with the inferred “shape”, and “childrenkwargs”.

`nubio.io.meta_mapper` (*vector*, *src*, *dst*, *map*)

Maps keys between source and destination dictionaries. Useful for explicit migration of data from one dataformat to another.

Arguments:

- `vector(nubox.char.CharArray)` a subclass of `nubox.char.CharArray`
- `src(str)` key to source metadata dictionary
- `dst(str)` key to destination metadata dictionary
- `map(sequence)` sequence of key mappings

`nubio.io.model_parser` (*fh*, *vector*, ***kwargs*)

Parses a file or file-like object and yield macromolecular models in a “vector” instance. Multiple models from a single file are supported.

Arguments:

- `fh(file)` file-like object open for reading.
- `vector(nubox.double.DoubleArray)` a subclass of `nubox.double.DoubleArray`

Additional keyworded arguments are passed to the “vector” constructor.

`nubio.io.seq_parser` (*fh*, *vector*, ***kwargs*)

Generator over sequences in a file-like object. Yields instances of the given “vector”.

Arguments:

- `fh(file)` A file-like object open for reading.
- `vector(nubox.char.CharVector)` a subclass of `nubox.char.CharVector`

Additional keyworded arguments are deep-copied and passed to the “vector” constructor.

2.6.10 nubio.io.fasta

Provides read and write support for the fasta file format.

`nubio.io.fasta.fasta_aln_parser` (*fh*)

Parses a fasta file with multiple sequences and generates two lists one of the “sequences” and one of “meta” associated with the “sequences”. The “sequences” are bytes. The “meta” dictionary corresponds to the “meta” argument of the default NuBox constructor. Information specific to a fasta file are stored under the “fasta” key. Currently “comments”, and “header” keys are created.

Arguments:

- `fh(file)` an open for reading file-like with fasta content, a sequence of lines or a `StringIO` object.

`nubio.io.fasta.fasta_aln_writer` (*fh*, *vector*, *width=80*, ***kwargs*)

Writes a `SymbolArray` instance or a (sequences, metadata) tuple to an open for writing file handle in the fasta file format. Supports other objects with a `write` method. The “width” of the sequence can be specified. Sequences should be valid `fasta_seq_writer` vectors. Does not support alignment metadata. Only supports sequences comments. Returns `None`.

Arguments:

- `fh(file)` An open for writing file handle or object with a `write` method.
- `vector(CharVector instance or sequence)` The contents of the “sequence” should be (sequences, metadata), where sequences is a valid input for `fasta_seq_writer`. And metadata is the (currently) ignored per-alignment metadata.

Additional keyworded arguments are passed to the `textwrap.wrap` function.

`nubio.io.fasta.fasta_seq_parser` (*fh*)

Parses a fasta file and generates tuples of (“sequence”, “meta”) for each sequence in the file. This parser does not preserve newlines in a fasta file. The “sequence” is returned as bytes, “meta” is a dict.

If you want to create a `CharVector` (or sub-class thereof) instance you have to supply “sequence” and “meta” to its constructor `CharVector(*fasta_seq_parser(fh))`.

The “meta” dictionary corresponds to the “meta” argument of the default `NuBox` constructor. Information specific to a fasta file are stored under the “fasta” key. Currently “comments”, and “header” keys are created.

Arguments:

- `fh(file)` A file-like object open for reading with “fasta” content, a sequence of lines is also valid.

`nubio.io.fasta.fasta_seq_writer(fh, vector, width=80, **kwargs)`

Writes a `CharVector` (or sub-class thereof) instance or a (“sequence”, “meta”) tuple to an open for writing file in the fasta file format. Supports other file-like objects with a `write` method. The width of the sequence (not comments) can be adjusted with the “width” argument. The “meta” dictionary or `vector.meta` property must include `meta["fasta"]["header"]`.

The “meta” dictionary should corresponds to the “meta” argument of the default `NuBox` constructor. Information specific to a fasta file are stored under the “fasta” key. Currently “comments”, and “header” keys are created.

Arguments:

- `fh(file or file-like)` an open for writing file handle or file-like object with a `write` method.
- `vector(CharVector or sequence)` if sequence the contents should be “letters” and “meta”, where “letters” is a sequence of bytes and comments is a dict.

Additional keyworded arguments are passed to the `textwrap.wrap` function.

`nubio.io.fasta.fasta_tmp(vector, **kwargs)`

Writes sequence(s) to a temporary file. Returns a file handle open for reading and writing. If the handle is closed or garbage collected the temporary file will be deleted from the file system. Contents of the file will be in the fasta file format.

Arguments:

- `vector(CharVector instance or sequence)`. This should be a valid input for `fasta_writer` or a `CharArray` if multiple sequences are to be written to one file.

Additional keyworded arguments are passed to `fasta_seq_writer`.

2.6.11 nubio.io.stockholm

Provides read and write support for the stockholm file format.

`nubio.io.stockholm.stockholm_aln_parser(fh)`

Parses a file or file-like object with contents in the stockholm file format returns a tuple of sequences, sequence metadata and alignment metadata. To parse directly into a `nubox` or `nubio` data container use `nubio.io.parse_aln`.

`nubio.io.stockholm.stockholm_aln_writer(fh, vector, feats=(), width=20)`

Writes a file or file-like object in the stockholm file format from the supplied “vector”.

Arguments:

- `fh(file)` file-like object open for reading
- `vector(nubox.char.CharArray)` a subclass of `nubox.char.CharArray`
- `feats(tuple)` [default: ()] additional “meta” keys to be exported as sequence features `#=GS`.
- `width(int)` [default: 20] minimum size of the id field

`nubio.io.stockholm.stockholm_tmp` (*vector*, ***kwargs*)

Writes sequence alignment to a temporary file. Returns a file handle open for reading and writing. If the handle is closed or garbage collected the temporary file will be deleted from the file system. Contents of the file will be in the stockholm file format.

Arguments:

- `vector`(`CharArray` or iterable). This should be a valid input for `stockholm_aln_writer`.

Additional keyworded arguments are passed to `stockholm_aln_writer`.

2.6.12 nubio.io.pdb

Provides read/write support for the pdb file format.

`nubio.io.pdb.pdb_parser` (*fh*)

Parses a PDB file and for each model yields a tuple of (`atom_coordinates`, `atom_metadatas`, `model_metadata`). Coordinates are tuples of floats. The data associated with each “ATOM” or “HET-ATM” line in the PDB file is stored as `atom_metadata['pdb']`. Model metadata is a dictionary with PDB header data under the ‘pdb’ key in `model_metadata`.

Arguments:

- `fh(file)` A file-like object open for reading.

`nubio.io.pdb.pdb_tmp` (*vector*)

Writes a macromolecular model to a temporary PDB file. Returns a file handle open for reading and writing. If the handle is closed the temporary file will be (at some point) deleted from the file system.

Arguments

- `vector`(`Model` instance or sequence) see: `pdb_writer`

`nubio.io.pdb.pdb_writer` (*fh*, *vector*)

Writes a PDB file given the contents of a sequence of atom coordinates, atom metadata dictionaries and a model metadata dictionary. The “vector” can be either a tuple of (`atom_coordinates`, `atom_metadatas`, `model_metadata`) or an instance of `Model`.

Arguments:

- `fh(file)` An open for writing file handle or object with `write` and `writelines` methods.
- `vector`(sequence or `Model` instance) This should be either a sequence of atom coordinates, metadata and model metadata or a `nubio.core.atom.Model` instance.

2.6.13 nubio.util

The `nubio.util` module provides various functions that do not fit anywhere else.

2.6.14 nubio.apps.app

Utility functions for creating and interacting with command-line applications.

`nubio.util.app.run_app` (*app*, *args=None*, *stdin=None*, ***kwargs*)

A simple wrapper around `subprocess.Popen` executes a command-line application with the given arguments and returns a 3-tuple of (`returncode`, `stdout`, `stderr`).

Arguments:

- `app` (`str`) The name of the application to run.
- `args` (**sequence**) [**default: None**] A sequence of arguments to the program.
- `stdin` (`file`) A file-like object open for reading.

Additional keyworded arguments are passed to the `subprocess.Popen` constructor.

`nubio.util.app.sh` (*command*, ***kwargs*)

Executes `command`-line in the shell. No `stdin`, `stdout` and `stderr` go to `/dev/null` and the command is executed via the default shell. Additional keyworded arguments are passed to `Popen`.

Arguments:

- `command`(`str` or sequence of `str`) This command line will be executed. See the documentation for “args” in `subprocess.Popen`.

`nubio.util.app.tmp_file` (*filepath=None*, *overwrite=False*)

Creates a temporary file and returns a file handle. If “filepath” is given it will be created or overwritten if `overwrite` is `True`, else a temporary file will be created. The name of the file is accessible from the “name” attribute of the file handle, the file ceases to exist if it is closed or eventually disappears if the handle garbage collected.

Arguments:

- `filepath`(`str`) [**default: None**] Path to the file.
- `overwrite`(`bool`) [**default: False**] Should the destination be overwritten if it exists? (ignored if “filepath” is `None`)

2.6.15 nubio.util.url

Utility functions for using web-resources.

`nubio.util.url.restful` (*url*, *params*)

Retrieves a RESTful url via `urllib`

Arguments:

- `url`(`str`) URL-string
- `params`(`dict`) A dictionary of valid parameters for `urlencode`

INDICES AND TABLES

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

PYTHON MODULE INDEX

n

nubio.core, 6
nubio.core.aa, 6
nubio.core.atom, 7
nubio.core.nt, 6
nubio.data, 8
nubio.data.alphabet, 8
nubio.data.files, 9
nubio.data.score, 9
nubio.io, 9
nubio.io.fasta, 10
nubio.io.pdb, 12
nubio.io.stockholm, 11
nubio.util, 12
nubio.util.app, 12
nubio.util.url, 13

INDEX

A

Aa (class in nubio.core.aa), 6
AaAln (class in nubio.core.aa), 6
AaSeq (class in nubio.core.aa), 6
aln_parser() (in module nubio.io), 9
Atom (class in nubio.core.atom), 8
AtomData (class in nubio.core.atom), 8

C

clschild (nubio.core.aa.AaAln attribute), 6
clschild (nubio.core.aa.AaSeq attribute), 6
clschild (nubio.core.atom.Model attribute), 8
clschild (nubio.core.atom.ModelData attribute), 8
clschild (nubio.core.nt.CodonAln attribute), 7
clschild (nubio.core.nt.CodonSeq attribute), 7
clschild (nubio.core.nt.NtAln attribute), 7
clschild (nubio.core.nt.NtSeq attribute), 7
clsparent (nubio.core.aa.Aa attribute), 6
clsparent (nubio.core.aa.AaSeq attribute), 6
clsparent (nubio.core.nt.Codon attribute), 6
clsparent (nubio.core.nt.CodonSeq attribute), 7
clsparent (nubio.core.nt.Nt attribute), 7
clsparent (nubio.core.nt.NtSeq attribute), 7
Codon (class in nubio.core.nt), 6
CodonAln (class in nubio.core.nt), 7
CodonSeq (class in nubio.core.nt), 7
complement() (nubio.core.nt.Nt method), 7

E

exponent_score_matrix() (in module nubio.data.score), 9

F

fasta_aln_parser() (in module nubio.io.fasta), 10
fasta_aln_writer() (in module nubio.io.fasta), 10
fasta_seq_parser() (in module nubio.io.fasta), 10
fasta_seq_writer() (in module nubio.io.fasta), 11
fasta_tmp() (in module nubio.io.fasta), 11

M

meta_mapper() (in module nubio.io), 9
Model (class in nubio.core.atom), 8

model_parser() (in module nubio.io), 10
ModelData (class in nubio.core.atom), 8

N

Nt (class in nubio.core.nt), 7
NtAln (class in nubio.core.nt), 7
NtSeq (class in nubio.core.nt), 7
nubio.core (module), 6
nubio.core.aa (module), 6
nubio.core.atom (module), 7
nubio.core.nt (module), 6
nubio.data (module), 8
nubio.data.alphabet (module), 8
nubio.data.files (module), 9
nubio.data.score (module), 9
nubio.io (module), 9
nubio.io.fasta (module), 10
nubio.io.pdb (module), 12
nubio.io.stockholm (module), 11
nubio.util (module), 12
nubio.util.app (module), 12
nubio.util.url (module), 13

P

parse_blosum_out() (in module nubio.data.score), 9
parse_score_matrix() (in module nubio.data.score), 9
pdb_parser() (in module nubio.io.pdb), 12
pdb_tmp() (in module nubio.io.pdb), 12
pdb_writer() (in module nubio.io.pdb), 12

R

restful() (in module nubio.util.url), 13
run_app() (in module nubio.util.app), 12

S

seq_parser() (in module nubio.io), 10
sh() (in module nubio.util.app), 13
stockholm_aln_parser() (in module nubio.io.stockholm),
11
stockholm_aln_writer() (in module nubio.io.stockholm),
11

stockholm_tmp() (in module nubio.io.stockholm), 11

T

tmp_file() (in module nubio.util.app), 13

translate() (nubio.core.nt.Codon method), 7

NuBox Documentation

Release 1.0

July 30, 2010

CONTENTS

1	Contents	3
1.1	Data Containers	3
1.2	Utility Functions	3
1.3	Examples	4
1.4	NuBox API	4
2	Indices and tables	15
	Python Module Index	17
	Index	19

The `nubox` package provides a array-like data-structure to represent hierarchical data together with the associated metadata. The core of **NuBox** are two classes the `NdArray` and the `NuBox`, both from the `nubox.core` module. The `NdArray` is a very rudimentary, lightweight and pure-python replacement for `numpy.ndarray`. `NuBox` is a sub-class of `NdArray`, which adds support for hierarchical data i.e. boxes of boxes like atoms in a molecule.

The package is tested on Python 2.6+, Jython 2.5+

CONTENTS

1.1 Data Containers

NuBox provides two types of containers generic and typed. Typed `NuBox` instances are designed to hold data of a certain data-type. Generic arrays and boxes must have their dimensions and/or data-type defined at run-time.

1.1.1 Generic Containers

Generic data containers are

- `NdArray` a generic lightweight multidimensional array implementation.
- `NuBox` a generic hierarchical array-wrapper of arbitrary number of dimensions and data-types.
- `NuBoxScalar`, `NuBoxVector`, `NuBoxArray`, `NuBoxCube` a hierarchy of array-wrappers with a defined data-type, but defined dimensionality.

1.1.2 Typed `NuBox` sub-classes

The available typed subclasses of the **scalar**, **vector**, **array**, **cube** hierarchy are:

- `char nubox.char`
- `double nubox.double`
- `short nubox.short`
- `int nubox.int`
- `long nubox.long`

1.2 Utility Functions

Various utility functions are available from `nubox.util`. And include `list` and `tuple` manipulation, color conversion, data-type and typecode conversion.

1.3 Examples

Examples can be found in the `doc/examples` directory of the source distribution. See also `test/test_*.py` to explore the API.

1.4 NuBox API

1.4.1 `nubox.core`

Provides the `NdArray` class, a N-dimensional array the more fancy `NuBox` and the a basic hierarchy of sub-classes, which include `NuBoxScalar`, `NuBoxVector`, `NuBoxArray` and `NuBoxCube` which all are sub-classes of `NdArray` to represent hierarchical data and metadata.

class `nubox.core.NdArray` (*data, shape, typecode, store='array', copy=True*)

An N-dimensional **array** of N-dimensional **items**. An **array** is constructed from a iterable “data” container, a “shape” and a “typecode”. The “typecode” determines the type of data held in the **array** e.g. “l” for long or “o” for arbitrary object instances. The “shape” determines the number and lengths of the dimensions. Sub-arrays along the first dimension are called **children**. Sub-arrays along the next to last dimension are called **items**. The length of the last dimension of the **array** determines the dimension of the **item**. Arbitrary slices of the array are called **chunks**. Objects in the flattened **array** are called **elements**.

Sub-array naming convention:

- **array** refers to the whole `NdArray` instance
- **chunk** is and arbitrary multi-dimensional slice of an **array**
- **child** is a sub-array indexed along the *first* dimension
- **item** is a sub-array indexed along the *next to last* dimension
- **element** is an object of the flattened **array**

`NdArray` provides methods and syntactic sugar to access all the sub-array types conveniently:

- **chunk** via `array.get_chunk`, `array.set_chunk` or `array[<slice>]` returns: `NdArray`
- **child** via `array.get_child`, `array.set_child` or `array[<int>]` returns: `NdArray`
- **item** via `array.get_item` or `array.set_item` returns: `tuple`
- **element** via `array[<tuple>]` returns: `type`

An `NdArray` has the following properties, which are also mandatory constructor arguments.:

- “data” contains the raw data converted to container defined by “store”
- “shape” defines the number and length of **dimensions** of the array
- “typecode” defines the type of **elements** stored in “data”
- “store” defines the type of the data container

These properties are constructed from the following arguments:

- **data** (iterable or `None`) any iterable of **elements**
- **shape** (iterable) a tuple of `int`.
- **typecode** (`str`) a single character, must be a valid letter for `array.array` or “o” if “store” is “list”.

Additional arguments provide defaults:

- `store` ("array" or "list") [default: "array"] "array" corresponds to `array.array` and "list" to `list`.

- `copy` (bool) [default: True] If `False` the “data” will not be copied if it is in the same “store” type. Will raise a `ValueError` if `False`, but a copy is required.

dump (*filepath=None*)

Dumps a `NdArray` into a package.

Arguments:

- `filepath(str)` a file path with “.nubox” extension (will be erased if exists)

fill_item (*value*)

Modifies in-place the value of each **item** in the **array**.

Arguments:

- `value(sequence)` sequence of **item** size with **elements** of valid `type`

get_child (*index*)

Returns the **child** (sub-array) at “index” along the first dimension of `self`. Equivalent to `self[index]`.

Arguments:

- `index(int)` index along the first dimension

get_chunk (*slices*)

Returns a **chunk** (sub-array) i.e. the cartesian product of `slices` along the arrays dimensions.

Arguments:

- `slices(sequence)` a sequence of `slice` instances.

get_item (*i*)

Returns the *i*-th **item** at “index” in the flattened array. The result is a tuple of **item** size with elements of `type` corresponding to the array typecode.

Arguments:

- `i(int)` **item** number (index)

iter_child (*fast=False*)

Returns a `generator` over children (sub-arrays) along the first dimension of `self`. If “fast” is `True` all returned objects are the same `NdArray` instance with monkey-patched data.

Arguments

- `fast(bool)` If `True` only one instance of a `NdArray` is constructed.

iter_item ()

Returns a `generator` over **items** in the flattened array.

load (*filepath=None*)

Loads an `NdArray` from a package.

- `filepath(str)` a file path with “.nubox” extension

map_item (*function, *args, **kwargs*)

Applies a function in-place on the **items** of the flattened **array**.

Arguments:

- `function(callable)` will be applied on the **array items**.

Additional arguments and keyworded arguments are passed to the supplied `callable object`.

max_item()

Returns the largest **item** in the flattened **array**.

min_item()

Returns the smallest **item** in the flattened **array**.

new (*data=None, **kwargs*)

Creates a new instance of `NdArray` has a signature identical to the default constructor with the key difference that it by default clones all properties of the current instance. If no arguments are given this is identical to `deepcopy(self)`.

see: `NdArray.__init__`

Keyworded arguments are the same as for the `NuBox` constructor.

set_child (*index, value*)

Modifies in-place the data of the **child** (sub-array) at “index” along the first dimension of `self`. Equivalent to `self[index] = value`.

Arguments:

- `index(int)` index along the first dimension
- `value(sequence)` sequence of **elements** of length of **child**

set_chunk (*slices, value*)

Modifies in-place the **chunk** (sub-array) i.e. the cartesian product of slices along the array dimensions.

Argumentensts:

- `slices(sequence)` a sequence `slice` instances that should be of the same length as the “shape” of the **array**.
- `value(sequence)` a sequence of **elements** size of **chunk**

set_item (*i, value*)

Modifies in-place the *i*-th **item** in the flattened **array**.

Arguments:

- `i(int)` **item** number (index)
- `value(iterable)` sequence of **item** size with **elements** of valid `type`

class `nubox.core.NuBox` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)

The `NuBox` is an array object and sub-class of `NdArray`. The main addition is that it holds metadata, supports a hierarchy i.e. an **array** of sub-arrays. The main difference between `NuBox` and `NdArray` is in the constructor. Further the `NuBox` has additional properties (attributes).

Just like `NdArray`, `NuBox` provides methods and syntactic sugar to access all the sub-arrays conveniently:

- **chunk** via `array.get_chunk`, `array.set_chunk` or `array[<slice>]` returns: `self`
- **child** via `array.get_child`, `array.set_child` or `array[<int>]` returns: `self.child`
- **item** via `array.get_item` or `array.set_item` returns `tuple`
- **element** via `array[<tuple>]` returns: `type`

An `NuBox` has the following additional to `NdArray` proprties, which are also constructor arguments only “dummy” is strictly mandatory.:

- “meta” metadata for the current instance
- “childrenkwargs” a list, which defines the constructor arguments (keyworded arguments) of all children

- “parent” defines the NuBox sub-class of the parent instance
- “child” defines the NuBox sub-class of **child** instances
- “dummy” defines a prototype **item** of the array, must be compatible with the last dimension of “shape”

This translates to the following constructor elements:

- “meta” (dict) any dictionary which can be serialized
- “childrenkwargs” (list) a list of dict one for each **child** this is what is passed to the “child” constructor.
- “parent” (NuBox or None) A NuBox sub-class (not instance)
- “child” (NuBox or NOCHILD) A NuBox sub-class (not instance)
- “dummy” (tuple) a tuple of length equal to the last dimension of array

Additional properties (attributes) and arguments are the same as for ndarray.

get_chunk (*slices*)

Returns a **chunk** (sub-array) i.e. the cartesian product of *slices* along the arrays dimensions.

Arguments:

- slices*(iterable) an iterable of *slice* instances.

new (*data=None, **kwargs*)

Creates a new instance of NuBox with a signature identical to the default constructor with the key difference that it by default clones all properties of the current instance. If no arguments are given this is identical to `deepcopy(self)`.

Arguments:

- data* see: `NuBox.__init__`

Keyworded arguments are the same as for the NuBox constructor.

```
class nubox.core.NuBoxArray(data=None, meta=None, childrenkwargs=None, parent=None,
                          child=None, dummy=None, shape=None, typecode=None,
                          store=None)
```

A NuBoxArray has rank 3 and hold NuBoxVectors.

see: NuBox

clschild

alias of NuBoxVector

clsparent

alias of NuBoxCube

```
class nubox.core.NuBoxCube(data=None, meta=None, childrenkwargs=None, parent=None,
                          child=None, dummy=None, shape=None, typecode=None, store=None)
```

A NuBoxCube has rank 4 and hold NuBoxArrays.

see: NuBox

clschild

alias of NuBoxArray

```
class nubox.core.NuBoxScalar(data=None, meta=None, childrenkwargs=None, parent=None,
                             child=None, dummy=None, shape=None, typecode=None,
                             store=None)
```

A NuBoxScalar has rank 1 and has no **children**.

see: NuBox

clsparent

alias of `NuBoxVector`

class `nubox.core.NuBoxVector` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)

A `NuBoxVector` has rank 2 and hold `NuBoxScalars`.

see: `NuBox`

clschild

alias of `NuBoxScalar`

clsparent

alias of `NuBoxArray`

1.4.2 nubox.const

Provides constants used within `nubox`. For example `NAMED_COLORS`, `ARRAY_TYPECODES` and `NuBox` dummies `INT_DUMMY`, `FLOAT_DUMMY` `CHAR_DUMMY`.

1.4.3 nubox.shim

(internal) Provides compability between Python 2.6+, Jython 2.5+ and (not yet) Python 3.1+.

1.4.4 nubox.util

Provides utility functions used within `nubox`.

`nubox.util.checkitem` (*item*)

Determines **item** properties and returns bool values. Returns a tuple of (`isiterable`, `isarray`, `isarrayitem`, `islistitem`), where:

- `isatomic` is True if **item** is atomic i.e. not a container
- `isiterable` is True if **item** is an iterable
- `isarray` is True if **item** is an `array.array` instance
- **isarrayitem is True if item can be an array.array item**
- `islistitem` is True if **item** can be a list **item**

Arguments:

- `item(object)` an **item** to be placed in an `NdArray`

`nubox.util.checktypecode` (*typecode*)

Determines bool values of typecode properties returns a tuple of (`istc`, `isatc`, `isutc`, `isotc`, `isftc`, `isitc`), where:

- `istc` is any valid typecode
- `isatc` is an array typecode (valid for `array.array` instances)
- `isutc` is an unicode typecode
- `isotc` is an object typecode
- `isftc` is a float typecode

- `isitc` is an `int` or `long` typecode

Arguments:

- `typecode(str)` A single character identifying the typecode most likely one of `nubox.const.TYPECODES`

`nubox.util.compatypecodes` (*typecodes*)

Determines if an iterable of typecodes represents the same Python type. Returns a `bool`. Typecodes that resolve to the same type are “compatible” and generally can be stored in the same array without loss.

Arguments:

- `typecodes(sequence)` a sequence of typecodes from most likely from `nubox.const.TYPECODES`

`nubox.util.flatten` (*l*)

Flattens a list or tuple. Returns a list with all elements.

Arguments:

- `l(sequence)` A possibly nested list or tuple of elements

`nubox.util.gettype` (*item*)

Determines the type and typecode of an **item**. Returns a tuple with a Python type or class and itemcode str.

Arguments:

- `item(object)` an **item** to be placed in an `NdArray`

`nubox.util.hex_to_rgb` (*value*)

Converts a hex string into a RGB tuple.

Arguments:

- `value(str)` A string representation of a hexadecimal number.

`nubox.util.make_colors` (*breakpoints, length=240*)

Creates a linear color map given RGB “breakpoints”. Returns an array of colors of specified “length”. A breakpoint is any RGB color-tuple. By default returns an array valid for `hijack_colors`.

Arguments:

- `breakpoints (sequence)` A sequence of tuples of RGB values each RGB value is a floating point number from 0. to 1..
- `length(int)` [default: 240] The number of colors in the returned list

`nubox.util.rgb_to_hex` (*rgb*)

Converts a RGB sequence into a hex string.

Arguments:

- `rgb(sequence)` A sequence of `int` in the 0-255 range.

`nubox.util.slices_shape2indices_shape` (*slices, shape*)

Translates an iterable of `slice` instances and the corresponding “shape” i.e. lengths of each dimension into a tuple of indices and a new shape. This function de facto implements an nD-slice.

Arguments:

- `slices(iterable of slice)` a sequence of `slice` instances
- `shape(tuple)` a sequence of dimensions i.e. lengths for each of the slices.

`nubox.util.unnest` (*l*)

For a given iterable of iterables returns a list, which contains all sub-sequences in a depth-first manner. For example given a nested iterable: [(1, 2, 3), (3, (4, 5))] it returns: [[(1, 2, 3), (3, (4, 5))], (1, 2, 3), 1, 2, 3, (3, (4, 5)), 3, (4, 5), 4, 5].

Arguments:

- (sequence) A possibly nested list or tuple of elements

1.4.5 nubox.char

Provides the NuBox sub-class hierarchy to store characters i.e. bytes.

class `nubox.char.Char` (**args*, ***kwargs*)

A scalar NuBox to hold characters as bytes.

clsparent

alias of `CharVector`

fill_item (*value*)

Same as in parent class NuBox

get_item (*index*)

Same as in parent class NuBox

iter_item ()

Same as in parent class NuBox

recode (*code*, *inplace=True*)

This re-codes (translates) a CharX given a “code” dictionary. Returns a new instance if “inplace” is False.

Arguments:

- code(dict) Translation dictionary.

set_item (*index*, *value*)

Same as in parent class NuBox

str (*method*, **args*, ***kwargs*)

Applies a str method. If the method returns a str of the same length modifies data in-place, else returns the result. Involves one or two copies of the raw data.

Arguments:

- method(str) Any method of a str instance e.g. “upper”

Additional arguments and keyworded arguments are passed to the called “method”.

tobytes ()

Returns the all data as a byte string.

class `nubox.char.CharArray` (**args*, ***kwargs*)

An array of Chars (vector of CharVectors).

clschild

alias of `CharVector`

clsparent

alias of `CharCube`

class `nubox.char.CharCube` (**args*, ***kwargs*)

A cube Chars (vector of CharArrays).

clschild
alias of `CharVector`

class `nubox.char.CharVector` (**args, **kwargs*)
A vector Chars.

clschild
alias of `Char`

clsparent
alias of `CharArray`

find (*ktup*)
Determines the start positions of all occurrences of *ktup* in *self*. The “*ktup*” argument should be a sequence of **children** data in the for of tuples.

Arguments:

- ktup*(sequence of **elements**) in the case of `Char` sub-classes elements are `str` of length 1

keep (*values*)
Returns a new instance, which keeps only **children** with **element** tuples in “*values*”. An element tuple is constructed from the whole “*data*” property of the **child**.

Arguments:

- values*(sequence of **element** tuples) a sequence of tuples. Each tuple should be the same length as a **child** and hold **elements**.

omit (*values*)
Returns a new instance, which omit **children** with **element** tuples in “*values*”. An element tuple is constructed from the whole “*data*” property of the **child**.

Arguments:

- values*(sequence of **element** tuples) a sequence of tuples. Each tuple should be the same length as a **child** and hold **elements**.

1.4.6 nubox.double

Provides the NuBox hierarchy of sub-classes to store floating point numbers as doubles.

class `nubox.double.Double` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
A scalar NuBox to hold a floating-point number as double.

clsparent
alias of `DoubleVector`

class `nubox.double.DoubleArray` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
An array of Doubles (vector of DoubleVectors).

clschild
alias of `DoubleVector`

clsparent
alias of `DoubleCube`

class `nubox.double.DoubleCube` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)

A cube of Doubles (vector of DoubleArrays).

clschild

alias of `DoubleArray`

class `nubox.double.DoubleVector` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)

A vector of Doubles.

clschild

alias of `Double`

clsparent

alias of `DoubleArray`

1.4.7 nubox.int

Provides the NuBox hierarchy of sub-classes to store integer numbers as ints.

class `nubox.int.Int` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)

A scalar NuBox to hold an integer number as an int.

clsparent

alias of `IntVector`

class `nubox.int.IntArray` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)

An array of Ints (vector of IntVectors).

clschild

alias of `IntVector`

clsparent

alias of `IntCube`

class `nubox.int.IntCube` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)

A cube of Ints (vector of IntArrays).

clschild

alias of `IntArray`

class `nubox.int.IntVector` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)

A vector of Ints.

clschild

alias of `Int`

clsparent

alias of `IntArray`

1.4.8 nubox.short

Provides the NuBox hierarchy of sub-classes to store integer numbers as longs.

class `nubox.short.Short` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
 A scalar NuBox to hold an integer number as short.

clsparent
 alias of `ShortVector`

class `nubox.short.ShortArray` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
 An array of `Shorts` (vector of `ShortVectors`).

clschild
 alias of `ShortVector`

clsparent
 alias of `ShortCube`

class `nubox.short.ShortCube` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
 A cube of `Shorts` (vector of `ShortArrays`).

clschild
 alias of `ShortArray`

class `nubox.short.ShortVector` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
 A vector of `Shorts`.

clschild
 alias of `Short`

clsparent
 alias of `ShortArray`

1.4.9 nubox.long

Provides the NuBox hierarchy of sub-classes to store integer numbers as longs.

class `nubox.long.Long` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
 A scalar NuBox to hold an integer number as long.

clsparent
 alias of `LongVector`

class `nubox.long.LongArray` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
 An array of `Longs` (vector of `LongVectors`).

clschild
 alias of `LongVector`

clsparent
 alias of `LongCube`

class `nubox.long.LongCube` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)
 A cube of `Longs` (vector of `LongArrays`).

clschild

alias of `LongArray`

class `nubox.long.LongVector` (*data=None, meta=None, childrenkwargs=None, parent=None, child=None, dummy=None, shape=None, typecode=None, store=None*)

A vector of Longs.

clschild

alias of `Long`

clsparent

alias of `LongArray`

INDICES AND TABLES

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

PYTHON MODULE INDEX

n

- nubox.char, 10
- nubox.const, 8
- nubox.core, 4
- nubox.double, 11
- nubox.int, 12
- nubox.long, 13
- nubox.shim, 8
- nubox.short, 12
- nubox.util, 8

INDEX

C

Char (class in nubox.char), 10
CharArray (class in nubox.char), 10
CharCube (class in nubox.char), 10
CharVector (class in nubox.char), 11
checkitem() (in module nubox.util), 8
checktypecode() (in module nubox.util), 8
clschild (nubox.char.CharArray attribute), 10
clschild (nubox.char.CharCube attribute), 10
clschild (nubox.char.CharVector attribute), 11
clschild (nubox.core.NuBoxArray attribute), 7
clschild (nubox.core.NuBoxCube attribute), 7
clschild (nubox.core.NuBoxVector attribute), 8
clschild (nubox.double.DoubleArray attribute), 11
clschild (nubox.double.DoubleCube attribute), 12
clschild (nubox.double.DoubleVector attribute), 12
clschild (nubox.int.IntArray attribute), 12
clschild (nubox.int.IntCube attribute), 12
clschild (nubox.int.IntVector attribute), 12
clschild (nubox.long.LongArray attribute), 13
clschild (nubox.long.LongCube attribute), 13
clschild (nubox.long.LongVector attribute), 14
clschild (nubox.short.ShortArray attribute), 13
clschild (nubox.short.ShortCube attribute), 13
clschild (nubox.short.ShortVector attribute), 13
clsparent (nubox.char.Char attribute), 10
clsparent (nubox.char.CharArray attribute), 10
clsparent (nubox.char.CharVector attribute), 11
clsparent (nubox.core.NuBoxArray attribute), 7
clsparent (nubox.core.NuBoxScalar attribute), 7
clsparent (nubox.core.NuBoxVector attribute), 8
clsparent (nubox.double.Double attribute), 11
clsparent (nubox.double.DoubleArray attribute), 11
clsparent (nubox.double.DoubleVector attribute), 12
clsparent (nubox.int.Int attribute), 12
clsparent (nubox.int.IntArray attribute), 12
clsparent (nubox.int.IntVector attribute), 12
clsparent (nubox.long.Long attribute), 13
clsparent (nubox.long.LongArray attribute), 13
clsparent (nubox.long.LongVector attribute), 14
clsparent (nubox.short.Short attribute), 13

clsparent (nubox.short.ShortArray attribute), 13
clsparent (nubox.short.ShortVector attribute), 13
compatypecodes() (in module nubox.util), 9

D

Double (class in nubox.double), 11
DoubleArray (class in nubox.double), 11
DoubleCube (class in nubox.double), 11
DoubleVector (class in nubox.double), 12
dump() (nubox.core.NdArray method), 5

F

fill_item() (nubox.char.Char method), 10
fill_item() (nubox.core.NdArray method), 5
find() (nubox.char.CharVector method), 11
flatten() (in module nubox.util), 9

G

get_child() (nubox.core.NdArray method), 5
get_chunk() (nubox.core.NdArray method), 5
get_chunk() (nubox.core.NuBox method), 7
get_item() (nubox.char.Char method), 10
get_item() (nubox.core.NdArray method), 5
gettype() (in module nubox.util), 9

H

hex_to_rgb() (in module nubox.util), 9

I

Int (class in nubox.int), 12
IntArray (class in nubox.int), 12
IntCube (class in nubox.int), 12
IntVector (class in nubox.int), 12
iter_child() (nubox.core.NdArray method), 5
iter_item() (nubox.char.Char method), 10
iter_item() (nubox.core.NdArray method), 5

K

keep() (nubox.char.CharVector method), 11

L

load() (nubox.core.NdArray method), 5
Long (class in nubox.long), 13
LongArray (class in nubox.long), 13
LongCube (class in nubox.long), 13
LongVector (class in nubox.long), 14

M

make_colors() (in module nubox.util), 9
map_item() (nubox.core.NdArray method), 5
max_item() (nubox.core.NdArray method), 5
min_item() (nubox.core.NdArray method), 6

N

NdArray (class in nubox.core), 4
new() (nubox.core.NdArray method), 6
new() (nubox.core.NuBox method), 7
NuBox (class in nubox.core), 6
nubox.char (module), 10
nubox.const (module), 8
nubox.core (module), 4
nubox.double (module), 11
nubox.int (module), 12
nubox.long (module), 13
nubox.shim (module), 8
nubox.short (module), 12
nubox.util (module), 8
NuBoxArray (class in nubox.core), 7
NuBoxCube (class in nubox.core), 7
NuBoxScalar (class in nubox.core), 7
NuBoxVector (class in nubox.core), 8

O

omit() (nubox.char.CharVector method), 11

R

recode() (nubox.char.Char method), 10
rgb_to_hex() (in module nubox.util), 9

S

set_child() (nubox.core.NdArray method), 6
set_chunk() (nubox.core.NdArray method), 6
set_item() (nubox.char.Char method), 10
set_item() (nubox.core.NdArray method), 6
Short (class in nubox.short), 12
ShortArray (class in nubox.short), 13
ShortCube (class in nubox.short), 13
ShortVector (class in nubox.short), 13
slices_shape2indices_shape() (in module nubox.util), 9
str() (nubox.char.Char method), 10

T

tobytes() (nubox.char.Char method), 10

U

unnest() (in module nubox.util), 9