Bonobo Documentation

Release 0.6.4

Romain Dorgueil

Apr 30, 2020

CONTENTS

CHAPTER

ONE

INSTALLATION

1.1 Create an ETL project

First, install the framework:

```
$ pip install --upgrade bonobo
```

Create a simple job:

```
$ bonobo init my-etl.py
```

And let's go for a test drive:

```
$ python my-etl.py
```

Congratulations, you ran your first Bonobo ETL job.

Now, you can head to *First steps*.

Note: It's often best to start with a single file then move it into a project (which, in python, needs to live in a package).

You can read more about this topic in the guide/packaging section, along with pointers on how to move this first file into an existing fully featured python package.

1.2 Other installation options

1.2.1 Install from PyPI

You can install it directly from the Python Package Index (like we did above).

```
$ pip install bonobo
```

To upgrade an existing installation, use *–upgrade*:

```
$ pip install --upgrade bonobo
```

1.2.2 Install from source

If you want to install an unreleased version, you can use git urls with pip. This is useful when using bonobo as a dependency of your code and you want to try a forked version of bonobo with your software. You can use a *git+http* string in your *requirements.txt* file. However, the best option for development on bonobo is an editable install (see below).

```
$ pip install git+https://github.com/python-bonobo/bonobo.git@develop#egg=bonobo
```

Note: Here, we use the *develop* branch, which is the incoming unreleased minor version. It's the way to "live on the edge", either to test your codebase with a future release, or to test unreleased features. You can use this technique to install any branch you want, and even a branch in your own repository.

1.2.3 Editable install

If you plan on making patches to Bonobo, you should install it as an "editable" package, which is a really great pip feature. Pip will clone your repository in a source directory and create a symlink for it in the site-package directory of your python interpreter.

```
$ pip install --editable git+https://github.com/python-bonobo/bonobo.git@develop
↪#egg=bonobo
```

Note: You can also use *-e*, the shorthand version of *–editable*.

Note: Once again, we use *develop* here. New features should go to *develop*, while bugfixes can go to *master*.

If you can't find the "source" directory, try running this:

```
$ python -c "import bonobo; print(bonobo.__path__)"
```

1.2.4 Local clone

Another option is to have a "local" editable install, which means you create the clone by yourself and make an editable install from the local clone.

```
$ git clone git@github.com:python-bonobo/bonobo.git
$ cd bonobo
$ pip install --editable .
```

You can develop on this clone, but you probably want to add your own repository if you want to push code back and make pull requests. I usually name the git remote for the main bonobo repository "upstream", and my own repository "origin".

```
$ git remote rename origin upstream
$ git remote add origin git@github.com:hartym/bonobo.git
$ git fetch --all
```

Of course, replace my github username by the one you used to fork bonobo. You should be good to go!

1.2.5 Preview versions

Sometimes, there are pre-versions available (before a major release, for example). By default, pip does not target pre-versions to avoid accidental upgrades to a potentially unstable version, but you can easily opt-in:

```
$ pip install --upgrade --pre bonobo
```

1.3 Supported platforms

1.3.1 Linux, OSX and other Unixes

Bonobo test suite runs continuously on Linux, and core developers use both OSX and Linux machines. Also, there are jobs running on production linux machines everyday, so the support for those platforms should be quite excellent.

If you're using some esoteric UNIX machine, there can be surprises (although we're not aware, yet). We do not support officially those platforms, but if you can actually fix the problems on those systems, we'll be glad to integrate your patches (as long as it is tested, for both existing linux environments and your strange systems).

1.3.2 Windows

Windows support is correct, as a few contributors helped us to test and fix the quirks.

There may still be minor issues on the windows platform, mostly due to the fact bonobo was not developed by windows users.

We're trying to look into that but energy available to provide serious support on windows is very limited.

If you have experience in this domain and you're willing to help, you're more than welcome!

CHAPTER

TWO

FIRST STEPS

Bonobo is an ETL (Extract-Transform-Load) framework for python 3.5. The goal is to define data-transformations, with python code in charge of handling similar shaped independent lines of data.

Bonobo *is not* a statistical or data-science tool. If you're looking for a data-analysis tool in python, use Pandas.

Bonobo is a lean manufacturing assembly line for data that let you focus on the actual work instead of the plumbery (execution contexts, parallelism, error handling, console output, logging, ...).

Bonobo uses simple python and should be quick and easy to learn.

Tutorials

2.1 Part 1: Let's get started!

To get started with **Bonobo**, you need to install it in a working python 3.5+ environment (you should use a virtualenv).

```
$ pip install bonobo
```

Check that the installation worked, and that you're using a version that matches this tutorial (written for bonobo v.0.6.4).

```
$ bonobo version
```

See *Installation* for more options.

2.1.1 Create an ETL job

Since Bonobo 0.6, it's easy to bootstrap a simple ETL job using just one file.

We'll start here, and the later stages of the tutorial will guide you toward refactoring this to a python package.

```
$ bonobo init tutorial.py
```

This will create a simple job in a *tutorial.py* file. Let's run it:

```
$ python tutorial.py
Hello
World
 - extract in=1 out=2 [done]
 - transform in=2 out=2 [done]
 - load in=2 [done]
```

Congratulations! You just ran your first **Bonobo** ETL job.

2.1.2 Inspect your graph

The basic building blocks of **Bonobo** are **transformations** and **graphs**.

Transformations are simple python callables (like functions) that handle a transformation step for a line of data.

Graphs are a set of transformations, with directional links between them to define the data-flow that will happen at runtime.

To inspect the graph of your first transformation:

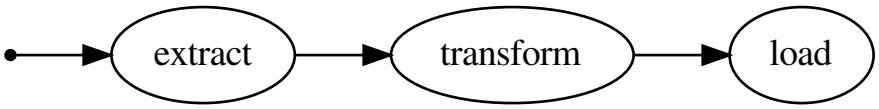
Note: You must install the graphviz software first. It is _not_ the python's graphviz package, you must install it using your system's package manager (apt, brew, ...).

For Windows users: you might need to add an entry to the Path environment variable for the *dot* command to be recognized

---

```
$ bonobo inspect --graph tutorial.py | dot -Tpng -o tutorial.png
```

Open the generated *tutorial.png* file to have a quick look at the graph.



You can easily understand here the structure of your graph. For such a simple graph, it's pretty much useless, but as you'll write more complex transformations, it will be helpful.

2.1.3 Read the Code

Before we write our own job, let's look at the code we have in *tutorial.py*.

Import

```
import bonobo
```

The highest level APIs of **Bonobo** are all contained within the top level **bonobo** namespace.

If you're a beginner with the library, stick to using only those APIs (they also are the most stable APIs).

If you're an advanced user (and you'll be one quite soon), you can safely use second level APIs.

The third level APIs are considered private, and you should not use them unless you're hacking on **Bonobo** directly.

Extract

```
def extract():
    yield 'hello'
    yield 'world'
```

This is a first transformation, written as a python generator, that will send some strings, one after the other, to its output.

Transformations that take no input and yields a variable number of outputs are usually called **extractors**. You'll encounter a few different types, either purely generating the data (like here), using an external service (a database, for example) or using some filesystem (which is considered an external service too).

Extractors do not need to have its input connected to anything, and will be called exactly once when the graph is executed.

Transform

```
def transform(*args):
    yield tuple(
        map(str.title, args)
    )
```

This is a second transformation. It will get called a bunch of times, once for each input row it gets, and apply some logic on the input to generate the output.

This is the most **generic** case. For each input row, you can generate zero, one or many lines of output for each line of input.

Load

```
def load(*args):
    print(*args)
```

This is the third and last transformation in our "hello world" example. It will apply some logic to each row, and have absolutely no output.

Transformations that take input and yields nothing are also called **loaders**. Like extractors, you'll encounter different types, to work with various external systems.

Please note that as a convenience mean and because the cost is marginal, most builtin *loaders* will send their inputs to their output unmodified, so you can easily chain more than one loader, or apply more transformations after a given loader.

Graph Factory

```
def get_graph(**options):
    graph = bonobo.Graph()
    graph.add_chain(extract, transform, load)
    return graph
```

All our transformations were defined above, but nothing ties them together, for now.

This "graph factory" function is in charge of the creation and configuration of a `bonobo.Graph` instance, that will be executed later.

By no mean is **Bonobo** limited to simple graphs like this one. You can add as many chains as you want, and each chain can contain as many nodes as you want.

Services Factory

```
def get_services(**options):
    return {}
```

This is the "services factory", that we'll use later to connect to external systems. Let's skip this one, for now.

(we'll dive into this topic in *Part 4: Services*)

Main Block

```
if __name__ == '__main__':
    parser = bonobo.get_argument_parser()
    with bonobo.parse_args(parser) as options:
        bonobo.run(
            get_graph(**options),
            services=get_services(**options)
        )
```

Here, the real thing happens.

Without diving into too much details for now, using the `bonobo.parse_args()` context manager will allow our job to be configurable, later, and although we don't really need it right now, it does not harm neither.

Note: This is intended to run in a console terminal. If you're working in a jupyter notebook, you need to adapt the thing to avoid trying to parse arguments, or you'll get into trouble.

2.1.4 Reading the output

Let's run this job once again:

```
$ python tutorial.py
Hello
World
 - extract in=1 out=2 [done]
 - transform in=2 out=2 [done]
 - load in=2 [done]
```

The console output contains two things.

- First, it contains the real output of your job (what was `print()`-ed to *sys.stdout*).
- Second, it displays the execution status (on *sys.stderr*). Each line contains a "status" character, the node name, numbers and a human readable status. This status will evolve in real time, and allows to understand a job's progress while it's running.
  - Status character:
    - " " means that the node was not yet started.
    - "-" means that the node finished its execution.

        * "+" means that the node is currently running.
        * "*!*" means that the node had problems running.
    - Numerical statistics:
        * "*in*=..." shows the input lines count, also known as the amount of calls to your transformation.
        * "*out*=..." shows the output lines count.
        * "*read*=..." shows the count of reads applied to an external system, if the transformation supports it.
        * "*write*=..." shows the count of writes applied to an external system, if the transformation supports it.
        * "*err*=..." shows the count of exceptions that happened while running the transformation. Note that exception will abort a call, but the execution will move to the next row.

However, if you run the tutorial.py it happens too fast and you can't see the status change. Let's add some delays to your code.

At the top of tutorial.py add a new import and add some delays to the 3 stages:

```
import time

def extract():
    """Placeholder, change, rename, remove... """
    time.sleep(5)
    yield 'hello'
    time.sleep(5)
    yield 'world'


def transform(*args):
    """Placeholder, change, rename, remove... """
    time.sleep(5)
    yield tuple(
        map(str.title, args)
    )


def load(*args):
    """Placeholder, change, rename, remove... """
    time.sleep(5)
    print(*args)
```

Now run tutorial.py again, and you can see the status change during the process.

2.1.5 Wrap up

That's all for this first step.

You now know:

- How to create a new job (using a single file).
- How to inspect the content of a job.
- What should go in a job file.
- How to execute a job file.
- How to read the console output.

It's now time to jump to *Part 2: Writing ETL Jobs*.

2.2 Part 2: Writing ETL Jobs

In **Bonobo**, an ETL job is a graph with some logic to execute it, like the file we created in the previous section.

You can learn more about the `bonobo.Graph` data-structure and its properties in the *graphs guide*.

2.2.1 Scenario

Let's create a sample application, which goal will be to integrate some data in various systems.

We'll use an open-data dataset, containing all the fablabs in the world.

We will normalize this data using a few different rules, then write it somewhere.

In this step, we'll focus on getting this data normalized and output to the console. In the next steps, we'll extend it to other targets, like files, and databases.

2.2.2 Setup

We'll change the *tutorial.py* file created in the last step to handle this new scenario.

First, let's remove all boilerplate code, so it looks like this:

```
import bonobo


def get_graph(**options):
    graph = bonobo.Graph()
    return graph


def get_services(**options):
    return {}


if __name__ == '__main__':
    parser = bonobo.get_argument_parser()
    with bonobo.parse_args(parser) as options:
        bonobo.run(get_graph(**options), services=get_services(**options))
```

Your job now contains the logic for executing an empty graph, and we'll complete this with our application logic.

2.2.3 Reading the source data

Let's add a simple chain to our *get_graph(...)* function, so that it reads from the fablabs open-data api.

The source dataset we'll use can be found on this site. It's licensed under *Public Domain*, which makes it just perfect for our example.

Note: There is a `bonobo.contrib.opendatasoft` module that makes reading from OpenDataSoft APIs easier, including pagination and limits, but for our tutorial, we'll avoid that and build it manually.

Let's write our extractor:

```
import requests

FABLABS_API_URL = 'https://public-us.opendatasoft.com/api/records/1.0/search/?
↪dataset=fablabs&rows=1000'

def extract_fablabs():
    yield from requests.get(FABLABS_API_URL).json().get('records')
```

This extractor will get called once, query the API url, parse it as JSON, and yield the items from the "records" list, one by one.

Note: You'll probably want to make it a bit more verbose in a real application, to handle all kind of errors that can happen here. What if the server is down? What if it returns a response which is not JSON? What if the data is not in the expected format?

For simplicity sake, we'll ignore that here but that's the kind of questions you should have in mind when writing pipelines.

To test our pipeline, let's use a `bonobo.Limit` and a `bonobo.PrettyPrinter`, and change our *get_graph(...)* function accordingly:

```
import bonobo

def get_graph(**options):
    graph = bonobo.Graph()
    graph.add_chain(
        extract_fablabs,
        bonobo.Limit(10),
        bonobo.PrettyPrinter(),
    )
    return graph
```

Running this job should output a bit of data, along with some statistics.

First, let's look at the statistics:

```
- extract_fablabs in=1 out=995 [done]
- Limit in=995 out=10 [done]
- PrettyPrinter in=10 out=10 [done]
```

It is important to understand that we extracted everything (995 rows), before droping 99% of the dataset.

This is OK for debugging, but not efficient.

Note: You should always try to limit the amount of data as early as possible, which often means not generating the data you won't need in the first place. Here, we could have used the *rows=* query parameter in the API URL to not request the data we would anyway drop.

2.2.4 Normalize

Warning: This section is missing. Sorry, but stay tuned! It'll be added soon.

2.2.5 Output

We used `bonobo.PrettyPrinter` to output the data.

It's a flexible transformation provided that helps you display the content of a stream, and you'll probably use it a lot for various reasons.

2.2.6 Moving forward

You now know:

- How to use a reader node.
- How to use the console output.
- How to limit the number of elements in a stream.
- How to pass data from one node to another.
- How to structure a graph using chains.

It's now time to jump to *Part 3: Working with Files*.

2.3 Part 3: Working with Files

Writing to the console is nice, but let's be serious, real world will require us to use files or external services.

Let's see how to use a few builtin writers and both local and remote filesystems.

2.3.1 Filesystems

In **Bonobo**, files are accessed within a **filesystem** service (a fs' FileSystem object).

As a default, you'll get an instance of a local filesystem mapped to the current working directory as the *fs* service. You'll learn more about services in the next step, but for now, let's just use it.

2.3.2 Writing to files

To write in a file, we'll need to have an open file handle available during the whole transformation life.

We'll use a context processor to do so. A context processor is something very much like a `contextlib.contextmanager`, that **Bonobo** will use to run a setup/teardown logic on objects that need to have the same lifecycle as a job execution.

Let's write one that just handle opening and closing the file:

```
def with_opened_file(self, context):
    with open('output.txt', 'w+') as f:
        yield f
```

Now, we need to write a *writer* transformation, and apply this context processor on it:

```
from bonobo.config import use_context_processor

@use_context_processor(with_opened_file)
def write_repr_to_file(f, *row):
    f.write(repr(row) + "\n")
```

The *f* parameter will contain the value yielded by the context processors, in order of appearance. You can chain multiple context processors. To find out about how to implement this, check the **Bonobo** guides in the documentation.

Please note that the `bonobo.config.use_context_processor()` decorator will modify the function in place, but won't modify its behaviour. If you want to call it out of the **Bonobo** job context, it's your responsibility to provide the right parameters (and here, the opened file).

To run this, change the last stage in the pipeline in get_graph to write_repr_to_file

```
def get_graph(**options):
    graph = bonobo.Graph()
    graph.add_chain(
        extract_fablabs,
        bonobo.Limit(10),
        write_repr_to_file,
    )
    return graph
```

Now run tutorial.py and check the output.txt file.

2.3.3 Using the filesystem

We opened the output file using a hardcoded filename and filesystem implementation. Writing flexible jobs include the ability to change the load targets at runtime, and **Bonobo** suggest to use the *fs* service to achieve this with files.

Let's rewrite our context processor to use it.

```
def with_opened_file(self, context):
    with context.get_service('fs').open('output.txt', 'w+') as f:
        yield f
```

The interface does not change much, but this small change allows the end-user to change the filesystem implementation at runtime, which is great for handling different environments (local development, staging servers, production, ...).

Note that **Bonobo** only provides very few services with default implementation (actually, only *fs* and *http*), but you can define all the services you want, depending on your system. You'll learn more about this in the next tutorial chapter.

2.3.4 Using a different filesystem

To change the *fs* implementation, you need to provide your implementation in the dict returned by *get_services()*.

Let's write to a remote location, which will be an Amazon S3 bucket. First, we need to install the driver:

```
pip install fs-s3fs
```

Then, just provide the correct bucket to *bonobo.open_fs()*:

```
def get_services(**options):
    return {
        'fs': bonobo.open_fs('s3://bonobo-examples')
    }
```

Note: You must provide a bucket for which you have the write permission, and it's up to you to setup your amazon credentials in such a way that *boto* can access your AWS account.

2.3.5 Using builtin writers

Until then, and to have a better understanding of what happens, we implemented our writers ourselves.

Bonobo contains writers for a variety of standard file formats, and you're probably better off using builtin writers.

Let's use a `bonobo.CsvWriter` instance instead, by replacing our custom transformation in the graph factory function:

```
def get_graph(**options):
    graph = bonobo.Graph()
    graph.add_chain(
        ...
        bonobo.CsvWriter('output.csv'),
    )
    return graph
```

2.3.6 Reading from files

Reading from files is done using the same logic as writing, except that you'll probably have only one call to a reader. You can read the file we just wrote by using a `bonobo.CsvReader` instance:

```
def get_graph(**options):
    graph = bonobo.Graph()
    graph.add_chain(
        bonobo.CsvReader('input.csv'),
        ...
    )
    return graph
```

2.3.7 Moving forward

You now know:

- How to use the filesystem (*fs*) service.
- How to read from files.
- How to write to files.
- How to substitute a service at runtime.

It's now time to jump to *Part 4: Services*.

2.4 Part 4: Services

All external dependencies (like filesystems, network clients, database connections, etc.) should be provided to transformations as a service. This will allow for great flexibility, including the ability to test your transformations isolated from the external world and easily switch to production (being user-friendly for people in system administration).

In the last section, we used the *fs* service to access filesystems, we'll go even further by switching our *requests* call to use the *http* service, so we can switch the *requests* session at runtime. We'll use it to add an http cache, which is a great thing to avoid hammering a remote API.

2.4.1 Default services

As a default, **Bonobo** provides only two services:

- *fs*, a `fs.osfs.OSFS` object to access files.
- *http*, a `requests.Session` object to access the Web.

2.4.2 Overriding services

You can override the default services, or define your own services, by providing a dictionary to the *services*= argument of *bonobo.run*. First, let's rewrite get_services:

```
import requests

def get_services():
    http = requests.Session()
    http.headers = {'User-Agent': 'Monkeys!'}
    return {
        'http': http
    }
```

2.4.3 Switching requests to use the service

Let's replace the `requests.get` call we used in the first steps to use the *http* service:

```
from bonobo.config import use

@use('http')
def extract_fablabs(http):
    yield from http.get(FABLABS_API_URL).json().get('records')
```

Tadaa, done! You're no more tied to a specific implementation, but to whatever `requests` -compatible object the user wants to provide.

2.4.4 Adding cache

Let's demonstrate the flexibility of this approach by adding some local cache for HTTP requests, to avoid hammering the API endpoint as we run our tests.

First, let's install *requests-cache*:

```
$ pip install requests-cache
```

Then, let's switch the implementation, conditionally.

```
def get_services(use_cache=False):
    if use_cache:
        from requests_cache import CachedSession
        http = CachedSession('http.cache')
    else:
        import requests
        http = requests.Session()

    return {
        'http': http
    }
```

Then in the main block, let's add support for a *–use-cache* argument:

```
if __name__ == '__main__':
    parser = bonobo.get_argument_parser()
    parser.add_argument('--use-cache', action='store_true', default=False)

    with bonobo.parse_args(parser) as options:
        bonobo.run(get_graph(**options), services=get_services(**options))
```

And you're done! Now, you can switch from using or not the cache using the *–use-cache* argument in command line when running your job.

2.4.5 Moving forward

You now know:

- How to use builtin service implementations
- How to override a service
- How to define your own service
- How to tune the default argument parser

It's now time to jump to *Part 5: Projects and Packaging*.

2.5 Part 5: Projects and Packaging

Throughout this tutorial, we have been working with one file managing a job but real life often involves more complicated setups, with relations and imports between different files.

Data processing is something a wide variety of tools may want to include, and thus **Bonobo** does not enforce any kind of project structure, as the target structure will be dictated by the hosting project. For example, a *pipelines* sub-package would perfectly fit a django or flask project, or even a regular package, but it's up to you to chose the structure of your project.

2.5.1 Imports mechanism

Bonobo does not enforce anything on how the python import mechanism work. Especially, it won't add anything to your *sys.path*, unlike some popular projects, because we're not sure that's something you want.

If you want to use imports, you should move your script into a python package, and it's up to you to have it setup correctly.

2.5.2 Moving into an existing project

First, and quite popular option, is to move your ETL job file into a package that already exists.

For example, it can be your existing software, eventually using some frameworks like django, flask, twisted, celery... Name yours!

We suggest, but nothing is compulsory, that you decide on a namespace that will hold all your ETL pipelines and move all your jobs in it. For example, it can be *mypkg.pipelines*.

2.5.3 Creating a brand new package

Because you may be starting a project involving some data-engineering, you may not have a python package yet. As it can be a bit tedious to setup right, there is a helper, using Medikit, that you can use to create a brand new project:

```
$ bonobo init --package pipelines
```

Answer a few questions, and you should now have a *pipelines* package, with an example transformation in it.

You can now follow the instructions on how to install it (*pip install –editable pipelines*), and the import mechanism will work "just right" in it.

2.5.4 Common stuff

Probably, you'll want to separate the *get_services()* factory from your pipelines, and just import it, as the dependencies may very well be project wide.

But hey, it's just python! You're at home, now!

2.5.5 Moving forward

That's the end of the tutorial, you should now be familiar with all the basics.

A few appendixes to the tutorial can explain how to integrate with other systems (we'll use the "fablabs" application created in this tutorial and extend it):

- *Working with Django*
- *Working with Docker*
- *Working with Jupyter*
- *Working with SQLAlchemy*

Then, you can either jump head-first into your code, or you can have a better grasp at all concepts by *reading the full bonobo guide*.

You should also join the slack community and ask all your questions there! No need to stay alone, and the only stupid question is the one nobody asks!

Happy data flows!

What's next?

Once you're familiar with all the base concepts, you can...

- Read the *Guides* to have a deep dive in each concept.
- Explore the *Extensions* to widen the possibilities:
  - *Working with Django*
  - *Working with Docker*
  - *Working with Jupyter*
  - *Working with SQLAlchemy*
- Open the *References* and start hacking like crazy.

You're not alone!

Good documentation is not easy to write.

Although all content here should be accurate, you may feel a lack of completeness, for which we plead guilty and apologize.

If you're stuck, please come to the Bonobo Slack Channel and we'll figure it out.

If you're not stuck but had trouble understanding something, please consider contributing to the docs (using GitHub pull requests).

> **Warning:** This section is being rewritten for **Bonobo** 0.6, and it's now in a "work in progress" state.
>
> You can read the tutorial for the previous version (0.5). Please note that things changed a bit since then and you'll have quirks here and there.
>
> You can also read the migration guide from 0.5 to 0.6 that will give you a good overview of the changes.
>
> Hopefully, this document will be updated soon, and please accept our apologies about this doc status until then.

CHAPTER

THREE

GUIDES

This section will guide you through your journey with Bonobo ETL.

3.1 Introduction

The first thing you need to understand before you use **Bonobo**, or not, is what it does and what it does not, so you can understand if it could be a good fit for your use cases.

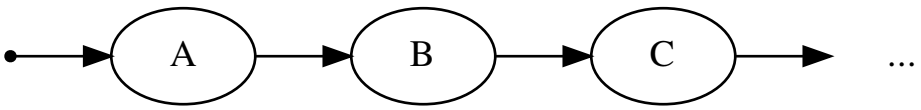
3.1.1 How it works?

Bonobo is an **Extract Transform Load** framework aimed at coders, hackers, or any other people who are at ease with terminals and source code files.

It is a **data streaming** solution, that treat datasets as ordered collections of independent rows, allowing to process them "first in, first out" using a set of transformations organized together in a directed graph.

Let's take a few examples.

Simplest linear graph



One of the simplest, by the book, cases, is an extractor sending to a transformation, itself sending to a loader (hence the "Extract Transform Load" name).

Note: Of course, **Bonobo** is aiming at real-world data transformations and can help you build all kinds of data-flows.

Bonobo will send an "impulsion" to all transformations linked to the *BEGIN* node (shown as a little black dot on the left).

On our example, the only node having its input linked to *BEGIN* is *A*.

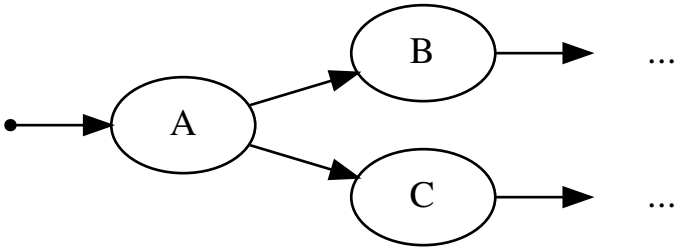
A's main topic will be to extract data from somewhere (a file, an endpoint, a database...) and generate some output. As soon as the first row of *A*'s output is available, **Bonobo** will start asking *B* to process it. As soon as the first row of *B*'s output is available, **Bonobo** will start asking *C* to process it.

While *B* and *C* are processing, *A* continues to generate data.

This approach can be efficient, depending on your requirements, because you may rely on a lot of services that may be long to answer or unreliable, and you don't have to handle optimizations, parallelism or retry logic by yourself.

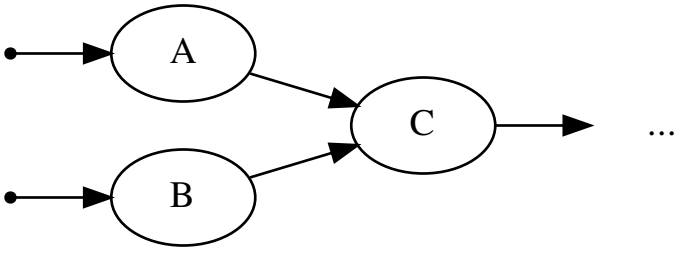
Note: The default execution strategy uses threads, and makes it efficient to work on I/O bound tasks. It's in the plans to have other execution strategies, based on subprocesses (for CPU-bound tasks) or *dask.distributed* (for big data tasks that requires a cluster of computers to process in reasonable time).

Graphs with divergence points (or forks)



In this case, any output row of *A*, will be **sent to both** *B* and *C* simultaneously. Again, *A* will continue its processing while *B* and *C* are working.

Graph with convergence points (or merges)



Now, we feed *C* with both *A* and *B* output. It is not a "join", or "cartesian product". It is just two different pipes plugged to *C* input, and whichever yields data will see this data feeded to *C*, one row at a time.

3.1.2 What is it not?

Bonobo is not:

- A data science, or statistical analysis tool, which need to treat the dataset as a whole and not as a collection of independent rows. If this is your need, you probably want to look at pandas.
- A workflow or scheduling solution for independent data-engineering tasks. If you're looking to manage your sets of data processing tasks as a whole, you probably want to look at Airflow. Although there is no **Bonobo** extension yet that handles that, it does make sense to integrate **Bonobo** jobs in an airflow (or other similar tool) workflow.
- A big data solution, as defined by Wikipedia. We're aiming at "small scale" data processing, which can be still quite huge for humans, but not for computers. If you don't know whether or not this is sufficient for your needs, it probably means you're not in "big data" land.

3.1.3 Where to jump next?

We suggest that you go through the *tutorial* first.

Then, you can read the guides, either using the order suggested or by picking the chapter that interest you the most at one given moment:

- *Introduction*
- *Transformations*
- *Graphs*
- *Services and dependencies*
- *Environment Variables*
- *Best Practices*
- *Debugging*
- *Plugins*

3.2 Transformations

Transformations are the smallest building blocks in **Bonobo**.

There is no special data-structure used to represent transformations, it's basically just a regular python callable, or even an iterable object (if it requires no input data).

Once in a graph, transformations become nodes and the data-flow between them is described using edges.

Note: In this chapter, we'll consider that anytime we need a "database", it's something we can get from the global namespace. This practice OK-ish for small jobs, but not at scale.

You'll learn in *Services and dependencies* how to manage external dependencies the right way.

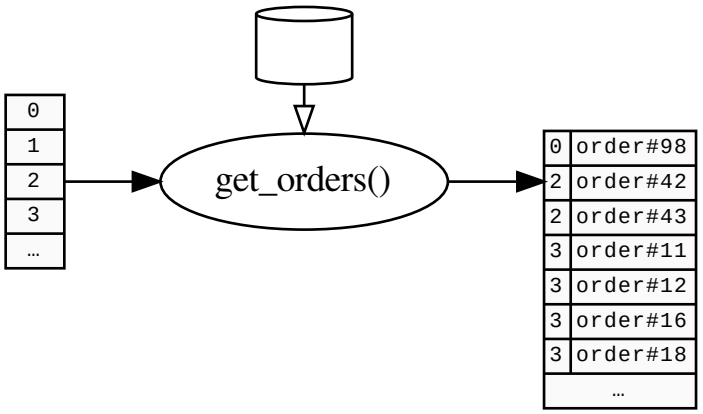
3.2.1 Transformation types

General case

The **general case** is a transformation that yields n outputs for each input.

You can implement it using a generator:

```
db = ...

def get_orders(user_id):
    for order in db.get_orders(user_id):
        yield user_id, order
```



Here, each row (containing a user id) will be transformed into a set of rows, each containing an user_id and an "order" object.

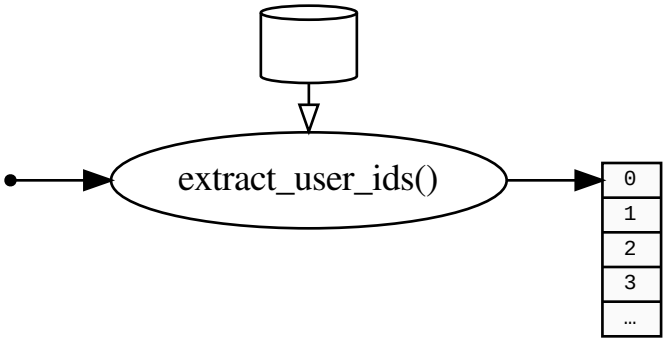
Extractor case

An **extractor** is a transformation that generates output without using any input. Usually, it does not generate this data out of nowhere, but instead connects to an external system (database, api, http, files . . .) to read the data from there.

It can be implemented in two different ways.

- You can implement it using a generator, like in the general case:

```
db = ...

def extract_user_ids():
    yield from db.select_all_user_ids()
```



- You can also use an iterator directly:

```
import bonobo

db = ...

def get_graph():
    graph = bonobo.Graph()
    graph.add_chain(
        db.select_all_user_ids(),
        ...
    )
    return graph
```

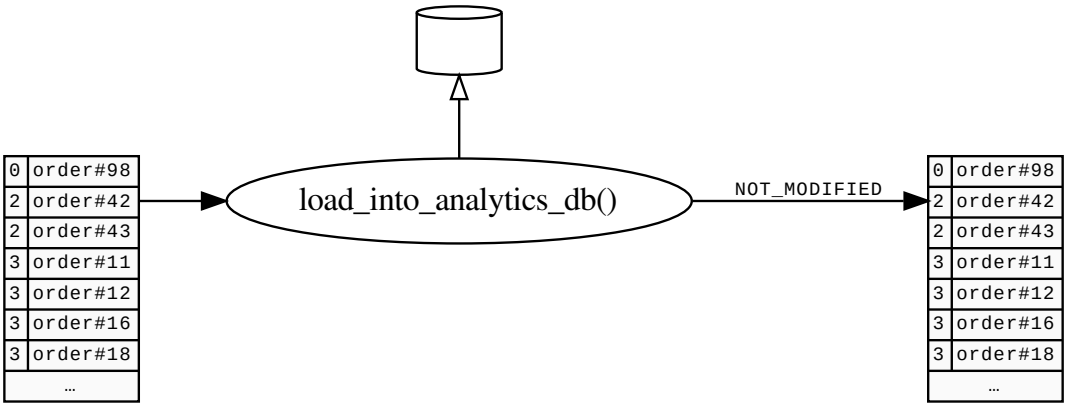
It is very convenient in many cases, when your existing system already have an interface that gives you iterators.

Note: It's important to use a generative approach that yield data as it is provided and not generate everything at once before returning, so **Bonobo** can pass the data to the next nodes as soon as it starts streaming.

Loader case

A **loader** is a transformation that sends its input into an external system. To have a perfect symmetry with extractors, we'd like not to have any output but as a convenience and because it has a negligible cost in **Bonobo**, the convention is that all loaders return *bonobo.constants.NOT_MODIFIED*, meaning that all rows that streamed into this node's input will also stream into its outputs, not modified. It allows to chain transformations even after a loader happened, and avoid using shenanigans to achieve the same thing:

```
from bonobo.constants import NOT_MODIFIED

analytics_db = ...

def load_into_analytics_db(user_id, order):
    analytics_db.insert_or_update_order(user_id, order['id'], order['amount'])
    return NOT_MODIFIED
```



3.2.2 Execution Context

Transformations being regular functions, a bit of machinery is required to use them as nodes in a streaming flow.

When a `bonobo.Graph` is executed, each node is wrapped in a `bonobo.execution.contexts.NodeExecutionContext` which is responsible for keeping the state of a node, within a given execution.

3.2.3 Inputs and Outputs

When run in an execution context, transformations have inputs and outputs, which means that **Bonobo** will pass data that comes in the input queue as calls, and push returned / yielded values into the output queue.

For thread-based strategies, the underlying implementation if the input and output queues is the standard `queue.Queue`.

Inputs

Todo: proofread, check consistency and correctness

All input is retrieved via the call arguments. Each line of input means one call to the callable provided. Arguments will be, in order:

- Injected dependencies (database, http, filesystem, ...)
- Position based arguments
- Keyword based arguments

You'll see below how to pass each of those.

Output

Todo: proofread, check consistency and correctness

Each callable can return/yield different things (all examples will use yield, but if there is only one output per input line, you can also return your output row and expect the exact same behaviour).

Todo: add rules for output parsing

The logic is defined in this piece of code, documentation will be added soon:

Listing 1: NodeExecutionContext._cast(self, _input, _output)

```
    def _cast(self, _input, _output):
        """
        Transforms a pair of input/output into the real slim shoutput.

        :param _input: Bag
        :param _output: mixed
        :return: Bag
        """

        if isenvelope(_output):
            _output, _flags, _options = _output.unfold()
        else:
            _flags, _options = [], {}

        if len(_flags):
            # TODO: parse flags to check constraints are respected (like not modified
↪alone, etc.)

            if F_NOT_MODIFIED in _flags:
                if self._output_type:
                    return ensure_tuple(_input, cls=self._output_type)
                return _input

            if F_INHERIT in _flags:
                if self._output_type is None:
```

(continues on next page)

(continued from previous page)

```
                    self._output_type = concat_types(
                        self._input_type, self._input_length, self._output_type, len(_
↪output)
                    )
                _output = _input + ensure_tuple(_output)

        if not self._output_type:
            if issubclass(type(_output), tuple):
                self._output_type = type(_output)

        return ensure_tuple(_output, cls=self._output_type)
```

Basically, after checking a few flags (*NOT_MODIFIED*, then *INHERIT*), it will "cast" the data into the "output type", which is either tuple or a kind of namedtuple.

Todo: document cast/input_type/output_type logic.

3.2.4 Class-based Transformations

For use cases that are either less simple or that requires better reusability, you may want to use classes to define some of your transformations.

Todo: narrative doc

See:

- `bonobo.config.Configurable`
- `bonobo.config.Option`
- `bonobo.config.Service`
- `bonobo.config.Method`
- `bonobo.config.ContextProcessor`

3.2.5 Naming conventions

The naming convention used is the following.

If you're naming something which is an actual transformation, that can be used directly as a graph node, then use underscores and lowercase names:

```
# instance of a class based transformation
filter = Filter(...)

# function based transformation
def uppercase(s: str) -> str:
    return s.upper()
```

If you're naming something which is configurable, that will need to be instantiated or called to obtain something that can be used as a graph node, then use camelcase names:

```
# configurable
class ChangeCase(Configurable):
    modifier = Option(default='upper')
    def __call__(self, s: str) -> str:
        return getattr(s, self.modifier)()

# transformation factory
def Apply(method):
    @functools.wraps(method)
    def apply(s: str) -> str:
        return method(s)
    return apply

# result is a graph node candidate
upper = Apply(str.upper)
```

3.2.6 Testing

As Bonobo use plain old python objects as transformations, it's very easy to unit test your transformations using your favourite testing framework. We're using pytest internally for Bonobo, but it's up to you to use the one you prefer.

If you want to test a transformation with the surrounding context provided (for example, service instances injected, and context processors applied), you can use `bonobo.execution.NodeExecutionContext` as a context processor and have bonobo send the data to your transformation.

```
from bonobo.execution import NodeExecutionContext

with NodeExecutionContext(
    JsonWriter(filename), services={'fs': ...}
) as context:
    # Write a list of rows, including BEGIN/END control messages.
    context.write_sync(
        {'foo': 'bar'},
        {'foo': 'baz'},
    )
```

3.2.7 Where to jump next?

We suggest that you go through the *tutorial* first.

Then, you can read the guides, either using the order suggested or by picking the chapter that interest you the most at one given moment:

- *Introduction*
- *Transformations*
- *Graphs*
- *Services and dependencies*
- *Environment Variables*
- *Best Practices*
- *Debugging*
- *Plugins*

3.3 Graphs

Graphs are the glue that ties transformations together. They are the only data-structure bonobo can execute directly. Graphs must be acyclic, and can contain as many nodes as your system can handle. However, although in theory the number of nodes can be rather high, practical cases usually do not exceed a few hundred nodes and even that is a rather high number you may not encounter so often.

Within a graph, each node are isolated and can only communicate using their input and output queues. For each input row, a given node will be called with the row passed as arguments. Each *return* or *yield* value will be put on the node's output queue, and the nodes connected in the graph will then be able to process it.

Bonobo is a line-by-line data stream processing solution.

Handling the data-flow this way brings the following properties:

- **First in, first out**: unless stated otherwise, each node will receeive the rows from FIFO queues, and so, the order of rows will be preserved. That is true for each single node, but please note that if you define "graph bubbles" (where a graph diverge in different branches then converge again), the convergence node will receive rows FIFO from each input queue, meaning that the order existing at the divergence point wont stay true at the convergence point.
- **Parallelism**: each node run in parallel (by default, using independent threads). This is useful as you don't have to worry about blocking calls. If a thread waits for, let's say, a database, or a network service, the other nodes will continue handling data, as long as they have input rows available.
- **Independence**: the rows are independent from each other, making this way of working with data flows good for line-by-line data processing, but also not ideal for "grouped" computations (where an output depends on more than one line of input data). You can overcome this with rolling windows if the input required are adjacent rows, but if you need to work on the whole dataset at once, you should consider other software.

Graphs are defined using `bonobo.Graph` instances, as seen in the previous tutorial step.

3.3.1 What can be used as a node?

TL;DR: ... anything, as long as it's callable() or iterable.

Functions

```
def get_item(id):
    return id, items.get(id)
```

When building your graph, you can simply add your function:

```
graph.add_chain(..., get_item, ...)
```

Or using the new syntax:

```
graph >> ... >> get_item >> ...
```

Note: Please note that we pass the function object, and not the result of the function being called. A common mistake is to call the function while building the graph, which won't work and may be tedious to debug.

As a convention, we use snake_cased objects when the object can be directly passed to a graph, like this function.

Some functions are factories for closures, and thus behave differently (as you need to call them to get an actual object usable as a transformation. When it is the case, we use CamelCase as a convention, as it behaves the same way as a class.

Classes

```
class Foo:
    ...

    def __call__(self, id):
        return id, self.get(id)
```

When building your graph, you can add an instance of your object (or even multiple instances, eventually configured differently):

```
graph.add_chain(..., Foo(), ...)
```

Or using the new syntax:

```
graph >> ... >> Foo() >> ...
```

Iterables (generators, lists, ...)

As a convenience tool, we can use iterables directly within a graph. It can either be used as producer nodes (nodes that are normally only called once and produce data) or, in case of generators, as transformations.

```
def product(x):
    for i in range(10)
        yield x, i, x * i
```

Then, add it to a graph:

```
graph.add_chain(range(10), product, ...)
```

Or using the new syntax:

```
graph >> range(10) >> product >> ...
```

Builtins

Again, as long as it is callable, you can use it as a node. It means that python builtins works (think about *print* or *str.upper*...)

```
graph.add_chain(range(ord("a"), ord("z")+1), chr, str.upper, print)
```

Or using the new syntax:

```
graph >> range(ord("a"), ord("z")+1) >> chr >> str.upper >> print
```

3.3.2 What happens during the graph execution?

Each node of a graph will be executed in isolation from the other nodes, and the data is passed from one node to the next using FIFO queues, managed by the framework. It's transparent to the end-user, though, and you'll only use function arguments (for inputs) and return/yield values (for outputs).

Each input row of a node will cause one call to this node's callable. Each output is cast internally as a tuple-like data structure (or more precisely, a namedtuple-like data structure), and for one given node, each output row must have the same structure.

If you return/yield something which is not a tuple, bonobo will create a tuple of one element.

Properties

Bonobo assists you with defining the data-flow of your data engineering process, and then streams data through your callable graphs.

- Each node call will process one row of data.
- Queues that flows the data between node are first-in, first-out (FIFO) standard python `queue.Queue`.
- Each node will run in parallel
- Default execution strategy use threading, and each node will run in a separate thread.

Fault tolerance

Node execution is fault tolerant.

If an exception is raised from a node call, then this node call will be aborted but bonobo will continue the execution with the next row (after outputing the stack trace and incrementing the "err" counter for the node context).

It allows to have ETL jobs that ignore faulty data and try their best to process the valid rows of a dataset.

Some errors are fatal, though.

If you pass a 2 elements tuple to a node that takes 3 args, **Bonobo** will raise an `bonobo.errors.UnrecoverableTypeError`, and exit the current graph execution as fast as it can (finishing the other node executions that are in progress first, but not starting new ones if there are remaining input rows).

3.3.3 Definitions

Graph

> A directed acyclic graph of transformations, that Bonobo can inspect and execute.

Node

> A transformation within a graph. The transformations are stateless, and have no idea whether or not they are included in a graph, multiple graph, or not at all.

3.3.4 Building graphs

Graphs in **Bonobo** are instances of `bonobo.Graph`

Graphs should be instances of `bonobo.Graph`. The `bonobo.Graph.add_chain()` method can take as many positional parameters as you want.

Note: As of **Bonobo** 0.7, a new syntax is available that we believe is more powerfull and more readable than the legacy *add_chain* method. The former API is here to stay and it's perfectly safe to use it (in fact, the new syntax uses *add_chain* under the hood).

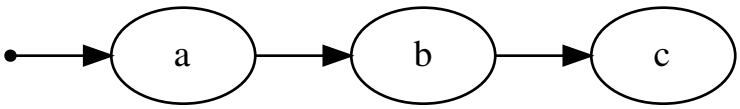
If it is an option for you, we suggest you consider the new syntax. During the transition period, we'll document both but the new syntax will eventually become default.

```
import bonobo

graph = bonobo.Graph()
graph.add_chain(a, b, c)
```

Or using the new syntax:

```
import bonobo

graph = bonobo.Graph()
graph >> a >> b >> c
```

Resulting graph:



3.3.5 Non-linear graphs

Divergences / forks

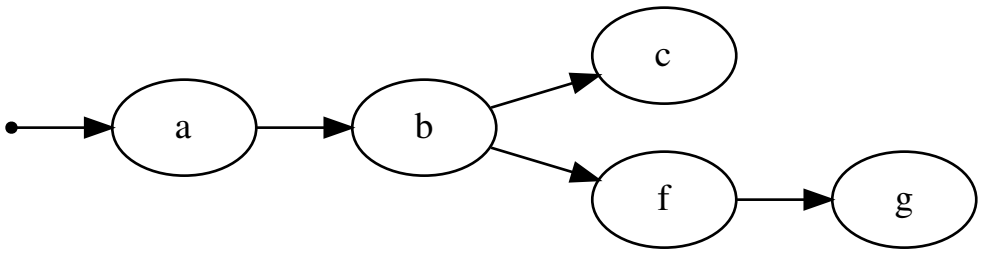
To create two or more divergent data streams ("forks"), you should specify the *_input* kwarg to *add_chain*.

```
import bonobo

graph = bonobo.Graph()
graph.add_chain(a, b, c)
graph.add_chain(f, g, _input=b)
```

Or using the new syntax:

```
import bonobo

graph = bonobo.Graph()
graph >> a >> b >> c
graph.get_cursor(b) >> f >> g
```

Resulting graph:



Note: Both branches will receive the same data and at the same time.

Convergence / merges

To merge two data streams, you can use the *_output* kwarg to *add_chain*, or use named nodes (see below).

```
import bonobo

graph = bonobo.Graph()

# Here we set _input to None, so normalize won't start on its own but only after it␣
↪receives input from the other chains.
graph.add_chain(normalize, store, _input=None)

# Add two different chains
graph.add_chain(a, b, _output=normalize)
graph.add_chain(f, g, _output=normalize)
```

Or using the new syntax:

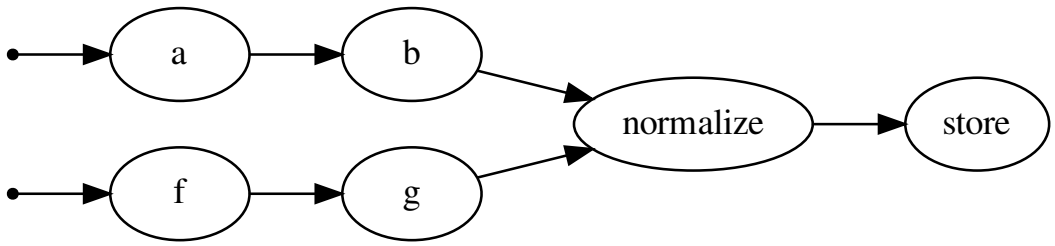
```
import bonobo

graph = bonobo.Graph()

# Here we set _input to None, so normalize won't start on its own but only after it␣
↪receives input from the other chains.
graph.get_cursor(None) >> normalize >> store

# Add two different chains
```

(continues on next page)

(continued from previous page)

```
graph >> a >> b >> normalize
graph >> f >> g >> normalize
```

Resulting graph:



Note: This is not a "join" or "cartesian product". Any data that comes from *b* or *g* will go through *normalize*, one at a time. Think of the graph edges as data flow pipes.

3.3.6 Named nodes

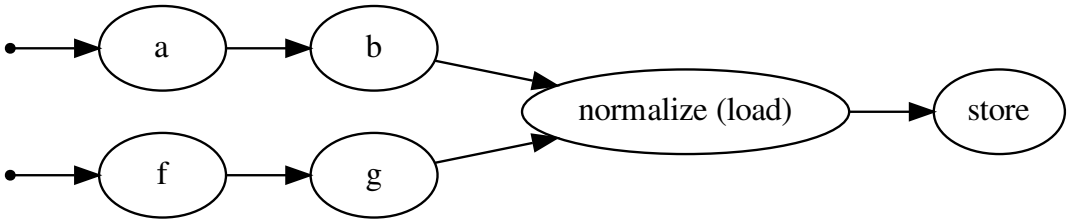
Using above code to create convergences often leads to code which is hard to read, because you have to define the "target" stream before the streams that logically goes to the beginning of the transformation graph. To overcome that, one can use "named" nodes.

Please note that naming a chain is exactly the same thing as naming the first node of a chain.

```
import bonobo

graph = bonobo.Graph()

# Here we mark _input to None, so normalize won't get the "begin" impulsion.
graph.add_chain(normalize, store, _input=None, _name="load")

# Add two different chains that will output to the "load" node
graph.add_chain(a, b, _output="load")
graph.add_chain(f, g, _output="load")
```

Using the new syntax, there should not be a need to name nodes. Let us know if you think otherwise by creating an issue.

Resulting graph:



You can also create single nodes, and the api provide the same capability on single nodes.

```
import bonobo

graph = bonobo.Graph()

# Create a node without any connection, name it.
graph.add_node(foo, _name="foo")

# Use it somewhere else as the data source.
graph.add_chain(..., _input="foo")

# ... or as the data sink.
graph.add_chain(..., _output="foo")
```

3.3.7 Orphan nodes / chains

The default behaviour of *add_chain* (or *get_cursor*) is to connect the first node to the special *BEGIN* token, which instruct **Bonobo** to call the connected node once without parameter to kickstart the data stream.

This is normally what you want, but there are ways to override it, as you may want to add "orphan" nodes or chains to your graph.

```
import bonobo

graph = bonobo.Graph()

# using add_node will naturally add a node as "orphan"
graph.add_node(a)

# using add_chain with "None" as the input will create an orphan chain
graph.add_chain(a, b, c, _input=None)

# using the new syntax, you can use either get_cursor(None) or the orphan() shortcut
graph.get_cursor(None) >> a >> b >> c

# ... using the shortcut ...
graph.orphan() >> a >> b >> c
```

3.3.8 Connecting two nodes

You may want to connect two nodes at some point. You can use *add_chain* without nodes to achieve it.

```
import bonobo

graph = bonobo.Graph()

# Create two "anonymous" nodes
graph.add_node(a)
graph.add_node(b)

# Connect them
graph.add_chain(_input=a, _output=b)
```

Or using the new syntax:

```
graph.get_cursor(a) >> b
```

3.3.9 Cursors

Cursors are simple structures that references a graph, a starting point and a finishing point. They can be used to manipulate graphs using the >> operator in an intuitive way.

To grab a cursor from a graph, you have different options:

```
# the most obvious way to get a cursor, its starting point will be "BEGIN"
cursor = graph.get_cursor()

# same thing, explicitely
cursor = graph.get_cursor(BEGIN)

# if you try to use a graph with the `>>` operator, it will create a cursor for you,␣
↪from "BEGIN"
cursor = graph >> ...  # same as `graph.get_cursor(BEGIN) >> ...`

# get a cursor pointing to nothing
cursor = graph.get_cursor(None)

# ... or in a more readable way
cursor = graph.orphan()
```

Once you get a cursor, you can use it to add nodes, concatenate it with othe cursors, etc. Everytime you call something that should result in a changed cursor, you'll get a new instance so your old cursor will still be available if you need it.

```
c1 = graph.orphan()

# append a node, get a new cursor
c2 = c1 >> node1

# create an orphan chain
c3 = graph.orphan() >> normalize

# concatenate a chain to an existing cursor
c4 = c2 >> c3
```

3.3.10 Inspecting graphs

Bonobo is bundled with an "inspector", that can use graphviz to let you visualize your graphs.

Read How to inspect and visualize your graph.

3.3.11 Executing graphs

There are two options to execute a graph (which have a similar result, but are targeting different use cases).

- You can use the bonobo command line interface, which is the highest level interface.
- You can use the python API, which is lower level but allows to use bonobo from within your own code (for example, a django management command).

Executing a graph with the command line interface

If there is no good reason not to, you should use *bonobo run ...* to run transformation graphs found in your python source code files.

```
$ bonobo run file.py
```

You can also run a python module:

```
$ bonobo run -m my.own.etlmod
```

In each case, bonobo's CLI will look for an instance of `bonobo.Graph` in your file/module, create the plumbing needed to execute it, and run it.

If you're in an interactive terminal context, it will use `bonobo.ext.console.ConsoleOutputPlugin` for display.

If you're in a jupyter notebook context, it will (try to) use `bonobo.ext.jupyter.JupyterOutputPlugin`.

Executing a graph using the internal API

To integrate bonobo executions in any other python code, you should use *`bonobo.run()`*. It behaves very similar to the CLI, and reading the source you should be able to figure out its usage quite easily.

3.3.12 Where to jump next?

We suggest that you go through the *tutorial* first.

Then, you can read the guides, either using the order suggested or by picking the chapter that interest you the most at one given moment:

- *Introduction*
- *Transformations*
- *Graphs*
- *Services and dependencies*
- *Environment Variables*
- *Best Practices*

- *Debugging*
- *Plugins*

3.4 Services and dependencies

You'll want to use external systems within your transformations, including databases, HTTP APIs, other web services, filesystems, etc.

Hardcoding those services is a good first step, but as your codebase grows, this approach will show its limits rather quickly.

- Hardcoded and tightly linked dependencies make your transformations hard to test, and hard to reuse.
- Processing data on your laptop is great, but being able to do it on different target systems (or stages), in different environments is more realistic. You'll want to configure a different database on a staging environment, pre-production environment, or production system. Maybe you have similar systems for different clients and want to select the system at runtime, etc.

Warning: This document is currently reviewed to check for correctness.

3.4.1 Definition of service dependencies

To solve this problem, we introduce a lightweight dependency injection system. It allows to define **named dependencies** in your transformations and provide an implementation at runtime.

For function-based transformations, you can use the `bonobo.config.use()` decorator to mark the dependencies. You'll still be able to call it manually, providing the implementation yourself, but in a bonobo execution context, it will be resolve and injected automatically, as long as you provided an implementation to the executor (more on that below).

```
from bonobo.config import use

@use('orders_database')
def select_all(database):
    yield from database.query('SELECT * FROM foo;')
```

For class based transformations, you can use `bonobo.config.Service`, a special descriptor (and subclass of `bonobo.config.Option`) that will hold the service names and act as a marker for runtime resolution of service instances.

```
from bonobo.config import Configurable, Service

class JoinDatabaseCategories(Configurable):
    database = Service('orders_database')

    def __call__(self, database, row):
        return {
            **row,
            'category': database.get_category_name_for_sku(row['sku'])
        }
```

Both of the above code samples tell bonobo that your transformation expects a service called "orders_database", which will be injected to your calls under the parameter name "database".

Providing implementations at run-time

Bonobo expects you to provide a dictionary of all service implementations required by your graph.

```
import bonobo

graph = bonobo.graph(...)

def get_services():
    return {
        'orders_database': my_database_service,
    }

if __name__ == '__main__':
    bonobo.run(graph, services=get_services())
```

Note: A dictionary, or dictionary-like, "services" named argument can be passed to the *bonobo.run()* API method. The "dictionary-like" part is the real keyword here. Bonobo is not a DIC library, and won't become one. So the implementation provided is pretty basic and feature-less. You can use much more involved libraries instead of the provided stub and, as long as it implements a dictionary-like interface, the system will use it.

The command line interface will look for services in two different places:

- A *get_services()* function present at the same level of your graph definition.
- A *get_services()* function in a *_services.py* file in the same directory as your graph's file, allowing to reuse the same service implementations for more than one graph.

Solving concurrency problems

If a service cannot be used by more than one thread at a time, either because it's just not threadsafe, or because it requires to carefully order the calls made (apis that includes nonces, or work on results returned by previous calls are usually good candidates), you can use the `bonobo.config.Exclusive` context processor to lock the use of a dependency for the time of the context manager (*with* statement)

```
from bonobo.config import Exclusive

def t1(api):
    with Exclusive(api):
        api.first_call()
        api.second_call()
        # ... etc
        api.last_call()
```

3.4.2 Read more

- See https://github.com/hartym/bonobo-sqlalchemy/blob/work-in-progress/bonobo_sqlalchemy/writers.py#L19 for example usage (work in progress).

3.4.3 Where to jump next?

We suggest that you go through the *tutorial* first.

Then, you can read the guides, either using the order suggested or by picking the chapter that interest you the most at one given moment:

- *Introduction*
- *Transformations*
- *Graphs*
- *Services and dependencies*
- *Environment Variables*
- *Best Practices*
- *Debugging*
- *Plugins*

3.5 Environment Variables

Best practice holds that variables should be passed to graphs via environment variables. Doing this is important for keeping sensitive data out of the code - such as an API token or username and password used to access a database. Not only is this approach more secure, it also makes graphs more flexible by allowing adjustments for a variety of environments and contexts. Importantly, environment variables are also the means by-which arguments can be passed to graphs.

Note: This document is about using your own settings and configuration values. If you're looking for bonobo's builtin settings, also configurable using environment variables, please check *Settings & Environment*.

3.5.1 Passing / Setting Environment Variables

Setting environment variables for your graphs to use can be done in a variety of ways and which one used can vary based-upon context. Perhaps the most immediate and simple way to set/override a variable for a given graph is simply to use the optional `--env` argument when running bonobo from the shell (bash, command prompt, etc). `--env` (or `-e` for short) should then be followed by the variable name and value using the syntax `VAR_NAME=VAR_VALUE`. Multiple environment variables can be passed by using multiple `--env` / `-e` flags (i.e. `bonobo run --env FIZZ=buzz ...` and `bonobo run --env FIZZ=buzz --env Foo=bar ...`). Additionally, in bash you can also set environment variables by listing those you wish to set before the *bonobo run* command with space separating the key-value pairs (i.e. `FIZZ=buzz bonobo run ...` or `FIZZ=buzz FOO=bar bonobo run ...`). Additionally, bonobo is able to pull environment variables from local '.env' files rather than having to pass each key-value pair individually at runtime. Importantly, a strict 'order of priority' is followed when setting environment variables so it is advisable to read and understand the order listed below to prevent

The order of priority is from lower to higher with the higher "winning" if set:

1. **default values** `os.getenv("VARNAME", default_value)` The user/writer/creator of the graph is responsible for setting these.
2. **`--default-env-file` values** Specify file to read default env values from. Each env var in the file is used if the var isn't already a corresponding value set at the system environment (system environment vars not overwritten).
3. **`--default-env` values** Works like #2 but the default `NAME=var` are passed individually, with one `key=value` pair for each `--default-env` flag rather than gathered from a specified file.
4. **system environment values** Env vars already set at the system level. It is worth noting that passed env vars via `NAME=value bonobo run ...` falls here in the order of priority.
5. **`--env-file` values** Env vars specified here are set like those in #2 albeit that these values have priority over those set at the system level.
6. **`--env` values** Env vars set using the `--env` / `-e` flag work like #3 but take priority over all other env vars.

3.5.2 Examples

The Examples below demonstrate setting one or multiple variables using both of these methods:

```
# Using one environment variable via a --env or --defualt-env flag:
bonobo run csvsanitizer --env SECRET_TOKEN=secret123
bonobo run csvsanitizer --defaul-env SECRET_TOKEN=secret123

# Using multiple environment variables via -e (env) and --default-env flags:
bonobo run csvsanitizer -e SRC_FILE=inventory.txt -e DST_FILE=inventory_processed.csv
bonobo run csvsanitizer --default-env SRC_FILE=inventory.txt --default-env DST_
↪FILE=inventory_processed.csv

# Using one environment variable inline (bash-like shells only):
SECRET_TOKEN=secret123 bonobo run csvsanitizer

# Using multiple environment variables inline (bash-like shells only):
SRC_FILE=inventory.txt DST_FILE=inventory_processed.csv bonobo run csvsanitizer

# Using an env file for default env values:
bonobo run csvsanitizer --default-env-file .env

# Using an env file for env values:
bonobo run csvsanitizer --env-file '.env.private'
```

3.5.3 ENV File Structure

The file structure for env files is incredibly simple. The only text in the file should be *NAME=value* pairs with one pair per line like the below.

```
# .env

DB_USER='bonobo'
DB_PASS='cicero'
```

3.5.4 Accessing Environment Variables from within the Graph Context

Environment variables, whether set globally or only for the scope of the graph, can be can be accessed using any of the normal means. It is important to note that whether set globally for the system or just for the graph context, environment variables are accessed by bonobo in the same way. In the example below the database user and password are accessed via the `os` module's `getenv` function and used to get data from the database.

```
import os

import bonobo
from bonobo.config import use


DB_USER = os.getenv('DB_USER')
DB_PASS = os.getenv('DB_PASS')


@use('database')
def extract(database):
    with database.connect(DB_USER, DB_PASS) as conn:
        yield from conn.query_all()


graph = bonobo.Graph(
    extract,
    bonobo.PrettyPrinter(),
)
```

3.5.5 Where to jump next?

We suggest that you go through the *tutorial* first.

Then, you can read the guides, either using the order suggested or by picking the chapter that interest you the most at one given moment:

- *Introduction*
- *Transformations*
- *Graphs*
- *Services and dependencies*
- *Environment Variables*
- *Best Practices*
- *Debugging*
- *Plugins*

3.6 Best Practices

Warning: This document needs to be rewritten for 0.6.

Especially, *Bag()* was removed, and **Bonobo** either ensure your i/o rows are tuples or some kind of namedtuples.

Please be aware of that while reading, and eventually check the migration guide to 0.6.

The nature of components, and how the data flow from one to another, can be a bit tricky. Hopefully, they should be very easy to write with a few hints.

3.6.1 Pure transformations

One "message" (a.k.a `bonobo.Bag` instance) may go through more than one component, and at the same time. To ensure your code is safe, one could `copy.copy()` each message on each transformation input but that's quite expensive, especially because it may not be needed.

Instead, we chose the opposite: copies are never made, instead you should not modify in place the inputs of your component before yielding them, which that mostly means that you want to recreate dicts and lists before yielding if their values changed.

Numeric values, strings and tuples being immutable in python, modifying a variable of one of those type will already return a different instance.

Examples will be shown with *return* statements, of course you can do the same with *yield* statements in generators.

Numbers

In python, numbers are immutable. So you can't be wrong with numbers. All of the following are correct.

```
def do_your_number_thing(n):
    return n

def do_your_number_thing(n):
    return n + 1

def do_your_number_thing(n):
    # correct, but bad style
    n += 1
    return n
```

The same is true with other numeric types, so don't be shy.

Tuples

Tuples are immutable, so you risk nothing.

```
def do_your_tuple_thing(t):
    return ('foo', ) + t

def do_your_tuple_thing(t):
    return t + ('bar', )
```

(continues on next page)

(continued from previous page)

```
def do_your_tuple_thing(t):
    # correct, but bad style
    t += ('baaaz', )
    return t
```

Strings

You know the drill, strings are immutable, too.

```
def do_your_str_thing(t):
    return 'foo ' + t + ' bar'

def do_your_str_thing(t):
    return ' '.join(('foo', t, 'bar', ))

def do_your_str_thing(t):
    return 'foo {} bar'.format(t)
```

You can, if you're using python 3.6+, use f-strings, but the core bonobo libraries won't use it to stay 3.5 compatible.

Dicts

So, now it gets interesting. Dicts are mutable. It means that you can mess things up if you're not cautious.

For example, doing the following may (will) cause unexpected problems:

```
def mutate_my_dict_like_crazy(d):
    # Bad! Don't do that!
    d.update({
        'foo': compute_something()
    })
    # Still bad! Don't mutate the dict!
    d['bar'] = compute_anotherthing()
    return d
```

The problem is easy to understand: as **Bonobo** won't make copies of your dict, the same dict will be passed along the transformation graph, and mutations will be seen in components downwards the output (and also upward). Let's see a more obvious example of something you should *not* do:

```
def mutate_my_dict_and_yield() -> dict:
    d = {}
    for i in range(100):
        # Bad! Don't do that!
        d['index'] = i
        yield d
```

Here, the same dict is yielded in each iteration, and its state when the next component in chain is called is undetermined (how many mutations happened since the *yield*? Hard to tell...).

Now let's see how to do it correctly:

```
def new_dicts_like_crazy(d):
    # Creating a new dict is correct.
    return {
```

(continues on next page)

(continued from previous page)

```
        **d,
        'foo': compute_something(),
        'bar': compute_anotherthing(),
    }

def new_dict_and_yield():
    for i in range(100):
        # Different dict each time.
        yield {
            'index': i
        }
```

I bet you think «Yeah, but if I create like millions of dicts . . . ».

Let's say we chose the opposite way and copied the dict outside the transformation (in fact, it's what we did in bonobo's ancestor). This means you will also create the same number of dicts, the difference is that you won't even notice it. Also, it means that if you want to yield the same dict 1 million times, going "pure" makes it efficient (you'll just yield the same object 1 million times) while going "copy crazy" would create 1 million identical objects.

Using dicts like this will create a lot of dicts, but also free them as soon as all the future components that take this dict as input are done. Also, one important thing to note is that most primitive data structures in python are immutable, so creating a new dict will of course create a new envelope, but the unchanged objects inside won't be duplicated.

Last thing, copies made in the "pure" approach are explicit, and usually, explicit is better than implicit.

3.6.2 Where to jump next?

We suggest that you go through the *tutorial* first.

Then, you can read the guides, either using the order suggested or by picking the chapter that interest you the most at one given moment:

- *Introduction*
- *Transformations*
- *Graphs*
- *Services and dependencies*
- *Environment Variables*
- *Best Practices*
- *Debugging*
- *Plugins*

3.7 Debugging

Note: This document writing is in progress, but its content should be correct (but succint).

3.7.1 Using a debugger (pdb...)

Using a debugger works (as in any python piece of code), but you must be aware that each node runs in a separate thread, which means a few things:

- If a breakpoint happens in a thread, then this thread will stop, but all other threads will continue running. This can be especially annoying if you try to use the pdb REPL for example, as your prompt will be overriden a few times/second by the current excution statistics.

  To avoid that, you can run bonobo with *QUIET=1* in environment, to hide statistics.

- If your breakpoint never happens (although it's at the very beginning of your transformation), it may mean that something happens out of the transform. The `bonobo.execution.contexts.NodeExecutionContext` instance that surrounds your transformation may be stuck in its *while True: transform()* loop.

  Break one level higher

3.7.2 Using printing statements

Of course, you can `print` things.

You can even add `print` statements in graphs, to `print` once per row.

A better `print` is available though, suitable for both flow-based data processing and human eyes. Check `bonobo.PrettyPrinter`.

3.7.3 Inspecting graphs

- Using the console: *bonobo inspect –graph*.
- Using Jupyter notebook: install the extension and just display a graph.

3.7.4 Where to jump next?

We suggest that you go through the *tutorial* first.

Then, you can read the guides, either using the order suggested or by picking the chapter that interest you the most at one given moment:

- *Introduction*
- *Transformations*
- *Graphs*
- *Services and dependencies*
- *Environment Variables*
- *Best Practices*
- *Debugging*
- *Plugins*

3.8 Plugins

3.8.1 Graph level plugins

3.8.2 Node level plugins

enhancers

node

-

3.8.3 Where to jump next?

We suggest that you go through the *tutorial* first.

Then, you can read the guides, either using the order suggested or by picking the chapter that interest you the most at one given moment:

- *Introduction*
- *Transformations*
- *Graphs*
- *Services and dependencies*
- *Environment Variables*
- *Best Practices*
- *Debugging*
- *Plugins*

CHAPTER

FOUR

EXTENSIONS

Extensions contains all things needed to work with a few popular third party tools.

Most of them are available as optional extra dependencies, and the maturity stage of each may vary.

4.1 Working with Django

Bonobo provides a lightweight integration with django, to allow to include ETL pipelines in your django management commands.

4.1.1 Quick start

To write a django management command that runs **Bonobo** job(s), just extend `ETLCommand` instead of `django.core.management.base.BaseCommand`, and override the `ETLCommand.get_graph()` method:

```
import bonobo
from bonobo.contrib.django import ETLCommand

class Command(ETLCommand):
    def get_graph(self, **options):
        graph = bonobo.Graph()
        graph.add_chain(...)
        return graph
```

Services

You can override `ETLCommand.get_services()` to provide your service implementations.

One common recipe to do so is to import it from somewhere else and override it as a `staticmethod`:

```
import bonobo
from bonobo.contrib.django import ETLCommand

from myproject.services import get_services

class Command(ETLCommand):
    get_services = staticmethod(get_services)

    def get_graph(...):
        ...
```

Multiple graphs

The `ETLCommand.get_graph()` method can also be implemented as a generator. In this case, each element yielded must be a graph, and each graph will be executed in order:

```
import bonobo
from bonobo.contrib.django import ETLCommand

class Command(ETLCommand):
    def get_graph(self, **options):
        yield bonobo.Graph(...)
        yield bonobo.Graph(...)
        yield bonobo.Graph(...)
```

This is especially helpful in two major cases:

- You must ensure that one job is finished before the next is run, and thus you can't add both graph's nodes in the same graph.
- You want to change which graph is run depending on command line arguments.

Command line arguments

Like with regular django management commands, you can add arguments to the argument parser by overriding `ETLCommand.add_arguments()`.

The only difference with django is that the provided argument parser will already have arguments added to handle environment.

4.1.2 Reference

`bonobo.contrib.django`

This module contains all tools for Bonobo and Django to interract nicely.

- `ETLCommand`
- `create_or_update()`

4.1.3 Source code

https://github.com/python-bonobo/bonobo/tree/master/bonobo/contrib/django

4.2 Working with Docker

Note: This extension is currently **BETA**.

Things will change, and although we use it on some real-world software, it may, or may not, satisfy your needs.

Read the introduction: https://www.bonobo-project.org/with/docker

4.2.1 Source code

https://github.com/python-bonobo/bonobo-docker

4.3 Working with Jupyter

Note: This extension is currently **BETA**.

Things will change, and although we use it on some real-world software, it may, or may not, satisfy your needs.

There is a builtin plugin that integrates (somewhat minimallistically, for now) bonobo within jupyter notebooks, so you can read the execution status of a graph within a nice (ok, not so nice) html/javascript widget.

4.3.1 Installation

Install *bonobo* with the **jupyter** extra:

```
pip install bonobo[jupyter]
```

Install the jupyter extension:

```
jupyter nbextension enable --py --sys-prefix widgetsnbextension
jupyter nbextension enable --py --sys-prefix bonobo.contrib.jupyter
```

4.3.2 Development

You should favor yarn over npm to install node packages. If you prefer to use npm, it's up to you to adapt the code.

To install the widget for development, make sure you're using an editable install of bonobo (see install document):

```
jupyter nbextension install --py --symlink --sys-prefix bonobo.contrib.jupyter
jupyter nbextension enable --py --sys-prefix bonobo.contrib.jupyter
```

If you want to change the javascript, you should run webpack in watch mode in some terminal:

```
cd bonobo/ext/jupyter/js
yarn install
./node_modules/.bin/webpack --watch
```

To compile the widget into a distributable version (which gets packaged on PyPI when a release is made), just run webpack:

```
./node_modules/.bin/webpack
```

4.3.3 Source code

https://github.com/python-bonobo/bonobo/tree/master/bonobo/contrib/jupyter

4.4 Working with Selenium

> **Warning:** This extension is currently **ALPHA**.
>
> Things will change, break, not work as expected, and the documentation is lacking some serious work.
>
> This section is here to give a brief overview but is neither complete nor definitive.
>
> You've been warned.

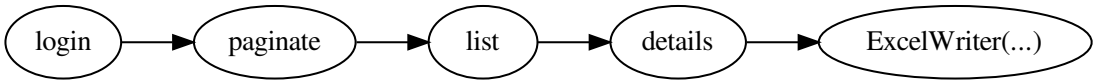
Writing web crawlers with Bonobo and Selenium is easy.

First, install **bonobo-selenium**:

```
$ pip install bonobo-selenium
```

The idea is to have one callable crawl one thing and delegate drill downs to callables further away in the chain.

An example chain could be:



Where each step would do the following:

- *login()* is in charge to open an authenticated session in the browser.
- *paginate()* open each page of a fictive list and pass it to next.
- *list()* take every list item and yield it.
- *details()* extract the data you're interested in.
- ... and the writer saves it somewhere.

4.4.1 Installation

4.4.2 Overview

4.4.3 Details

4.5 Working with SQLAlchemy

Note: This extension is currently **BETA**.

Things will change, and although we use it on some real-world software, it may, or may not, satisfy your needs.

Read the introduction: https://www.bonobo-project.org/with/sqlalchemy

4.5.1 Installation

To install the extension, use the *sqlalchemy* extra:

```
$ pip install bonobo[sqlalchemy]
```

Note: You can install more than one extra at a time separating the names with commas.

4.5.2 Overview and examples

First, you'll need a database connection (`sqlalchemy.engine.Engine` instance), that must be provided as a service.

```
import sqlalchemy

def get_services():
    return {
        'sqlalchemy.engine': sqlalchemy.create_engine(...)
    }
```

The *sqlalchemy.engine* name is the default name used by the provided transformations, but you can override it (for example if you need more than one connection) and specify the service name using *engine='myengine'* while building your transformations.

Lets create some tables and add some data. (You may need to edit the SQL if your database server uses a different version of SQL.)

```
CREATE TABLE test_in (
  id INTEGER PRIMARY KEY NOT NULL,
  text TEXT
);

CREATE TABLE test_out (
  id INTEGER PRIMARY KEY NOT NULL,
  text TEXT
);

INSERT INTO test_in (id, text) VALUES (1, 'Cat');
INSERT INTO test_in (id, text) VALUES (2, 'Dog');
```

There are two transformation classes provided by this extension.

One reader, one writer.

Let's select some data:

```
import bonobo
import bonobo_sqlalchemy
```

(continues on next page)

(continued from previous page)

```
def get_graph():
    graph = bonobo.Graph()
    graph.add_chain(
        bonobo_sqlalchemy.Select('SELECT * FROM test_in', limit=100),
        bonobo.PrettyPrinter(),
    )
    return graph
```

You should see:

```
$ python tutorial.py

| id[0] = 1
| text[1] = 'Cat'
|

| id[0] = 2
| text[1] = 'Dog'
|
 - Select in=1 out=2 [done]
 - PrettyPrinter in=2 out=2 [done]
```

Now let's insert some data:

```
import bonobo
import bonobo_sqlalchemy


def get_graph(**options):
    graph = bonobo.Graph()
    graph.add_chain(
        bonobo_sqlalchemy.Select('SELECT * FROM test_in', limit=100),
        bonobo_sqlalchemy.InsertOrUpdate('test_out')
    )

    return graph
```

If you check the *test_out* table, it should now have the data.

4.5.3 Reference

bonobo_sqlalchemy

4.5.4 Source code

https://github.com/python-bonobo/bonobo-sqlalchemy

CHAPTER

FIVE

REFERENCE

Reference documents of all stable APIs and modules. If something is not here, please be careful about using it as it means that the api is not yet 1.0-proof.

5.1 Bonobo

Module *bonobo*

Contains all the tools you need to get started with the framework, including (but not limited to) generic transformations, readers, writers, and tools for writing and executing graphs and jobs.

All objects in this module are considered very safe to use, and backward compatibility when moving up from one version to another is maximal.

5.1.1 Graphs

- `bonobo.structs.graphs.Graph`

5.1.2 Nodes

- `bonobo.nodes.CsvReader`
- `bonobo.nodes.CsvWriter`
- `bonobo.nodes.FileReader`
- `bonobo.nodes.FileWriter`
- `bonobo.nodes.Filter`
- `bonobo.nodes.FixedWindow`
- `bonobo.nodes.Format()`
- `bonobo.nodes.JsonReader`
- `bonobo.nodes.JsonWriter`
- `bonobo.nodes.LdjsonReader`
- `bonobo.nodes.LdjsonWriter`
- `bonobo.nodes.Limit`
- `bonobo.nodes.MapFields()`

- bonobo.nodes.OrderFields()
- bonobo.nodes.PickleReader
- bonobo.nodes.PickleWriter
- bonobo.nodes.PrettyPrinter
- bonobo.nodes.RateLimited
- bonobo.nodes.Rename()
- bonobo.nodes.SetFields()
- bonobo.nodes.Tee()
- bonobo.nodes.UnpackItems()
- bonobo.nodes.count()
- bonobo.nodes.identity()
- bonobo.nodes.noop()

5.1.3 Other top-level APIs

- bonobo.create_reader()
- *bonobo.create_strategy()*
- bonobo.create_writer()
- *bonobo.get_argument_parser()*
- *bonobo.get_examples_path()*
- *bonobo.inspect()*
- *bonobo.open_examples_fs()*
- *bonobo.open_fs()*
- *bonobo.parse_args()*
- *bonobo.run()*

create_reader

create_strategy

create_strategy(*name=None*)

Create a strategy, or just returns it if it's already one.

Parameters **name** –

Returns Strategy

create_writer

get_argument_parser

get_argument_parser(*parser=None*)

Creates an argument parser with arguments to override the system environment.

Api bonobo.get_argument_parser

Parameters **_parser** –

Returns

get_examples_path

get_examples_path(**pathsegments*)

inspect

inspect(*graph*, *, *plugins=None*, *services=None*, *strategy=None*, *format*)

open_examples_fs

open_examples_fs(**pathsegments*)

open_fs

open_fs(*fs_url=None*, **args*, ***kwargs*)

Wraps `fs.opener.registry.Registry.open_fs`, with default to local current working directory and expanding ~ in path.

Parameters

- **fs_url** (*str*) – A filesystem URL
- **parse_result** (`ParseResult`) – A parsed filesystem URL.
- **writeable** (*bool*) – True if the filesystem must be writeable.
- **create** (*bool*) – True if the filesystem should be created if it does not exist.
- **cwd** (*str*) – The current working directory (generally only relevant for OS filesystems).
- **default_protocol** (*str*) – The protocol to use if one is not supplied in the FS URL (defaults to `"osfs"`).

Returns `fs.base.FS` object

parse_args

parse_args(*mixed=None*)

Context manager to extract and apply environment related options from the provided argparser result.

A dictionnary with unknown options will be yielded, so the remaining options can be used by the caller.

Api bonobo.patch_environ

Parameters **mixed** – ArgumentParser instance, Namespace, or dict.

Returns

run

run(*graph*, *, *plugins=None*, *services=None*, *strategy=None*)

Main entry point of bonobo. It takes a graph and creates all the necessary plumbing around to execute it.

The only necessary argument is a `Graph` instance, containing the logic you actually want to execute.

By default, this graph will be executed using the "threadpool" strategy: each graph node will be wrapped in a thread, and executed in a loop until there is no more input to this node.

You can provide plugins factory objects in the plugins list, this function will add the necessary plugins for interactive console execution and jupyter notebook execution if it detects correctly that it runs in this context.

You'll probably want to provide a services dictionary mapping service names to service instances.

Parameters

- **graph** (*Graph*) – The `Graph` to execute.
- **strategy** (*str*) – The `bonobo.execution.strategies.base.Strategy` to use.
- **plugins** (*list*) – The list of plugins to enhance execution.
- **services** (*dict*) – The implementations of services this graph will use.

Return bonobo.execution.graph.GraphExecutionContext

5.2 Config

Module *bonobo.config*

The Config API, located under the *bonobo.config* namespace, contains all the tools you need to create configurable transformations, either class-based or function-based.

5.3 Constants

Module *bonobo.constants*

BEGIN

BEGIN token marks the entrypoint of graphs, and all extractors will be connected to this node.

Without this, it would be impossible for an execution to actually start anything, as it's the marker that tells **Bonobo** which node to actually call when the execution starts.

NOT_MODIFIED
: **NOT_MODIFIED** is a special value you can return or yield from a transformation to tell bonobo to reuse the input data as output.

  As a convention, all loaders should return this, so loaders can be chained.

EMPTY
: Shortcut for "empty tuple". It's often much more clear to write (especially in a test) *write(EMPTY)* than *write(())*, although strictly equivalent.

5.4 Execution

Module *bonobo.execution*

Execution logic, surrounding contexts for nodes and graphs and events.

This module is considered **internal**.

5.5 Nodes

Module *bonobo.nodes*

5.6 Graphs

Module *bonobo.structs.graphs*

5.7 Util

Module *bonobo.util*

The Util API, located under the *bonobo.util* namespace, contains helpers functions and decorators to work with and inspect transformations, graphs, and nodes.

5.8 Command-line

5.8.1 Bonobo Convert

Build a simple bonobo graph with one reader and one writer, then execute it, allowing to use bonobo in "no code" mode for simple file format conversions.

Syntax: *bonobo convert [-r reader] input_filename [-w writer] output_filename*

Todo: add a way to override default options of reader/writers, add a way to add "filters", for example this could be used to read from csv and write to csv too (or other format) but adding a geocoder filter that would add some fields.

5.8.2 Bonobo Inspect

Inspects a bonobo graph source files. For now, only support graphviz output.

Syntax: *bonobo inspect [–graph|-g] filename*

Requires graphviz if you want to generate an actual graph picture, although the command itself depends on nothing.

5.8.3 Bonobo Run

Run a transformation graph.

Syntax: *bonobo run [-c cmd | -m mod | file | -] [arg]*

Todo: implement -m, check if -c is of any use and if yes, implement it too. Implement args, too.

5.8.4 Bonobo RunC

Run a transformation graph in a docker container.

Syntax: *bonobo runc [-c cmd | -m mod | file | -] [arg]*

Todo: implement -m, check if -c is of any use and if yes, implement it too. Implement args, too.

Requires *bonobo-docker*, install with *docker* extra: *pip install bonobo[docker]*.

5.9 Settings & Environment

All settings that you can find in the `bonobo.settings` module. You can override those settings using environment variables. For you own settings and configuration values, see the *Environment Variables* guide.

5.9.1 Debug

Purpose Sets the debug mode, which is more verbose. Loglevel will be lowered to DEBUG instead of INFO.

Environment *DEBUG*

Setting *bonobo.settings.DEBUG*

Default *False*

5.9.2 Profile

Purpose Sets profiling, which adds memory/cpu usage output. Not yet fully implemented. It is expected that setting this to true will have a non-neglictible performance impact.

Environment *PROFILE*

Setting *bonobo.settings.PROFILE*

Default *False*

5.9.3 Quiet

Purpose Sets the quiet mode, which ask any output to be computer parsable. Formating will be removed, but it will allow to use unix pipes, etc. Not yet fully implemented, few transformations already use it. Probably, it should be the default on non-interactive terminals.

Environment *QUIET*

Setting *bonobo.settings.QUIET*

Default *False*

5.9.4 Logging Level

Purpose Sets the python minimum logging level.

Environment *LOGGING_LEVEL*

Setting *bonobo.settings.LOGGING_LEVEL*

Default *DEBUG* if DEBUG is False, otherwise *INFO*

Values *CRITICAL*, *FATAL*, *ERROR*, *WARNING*, *INFO*, *DEBUG*, *NOTSET*

5.9.5 I/O Format

Purpose Sets default input/output format for builtin transformations. It can be overriden on each node. The *kwargs* value means that each node will try to read its input from keywords arguments (and write similar formated output), while *arg0* means it will try to read its input from the first positional argument (and write similar formated output).

Environment *IOFORMAT*

Setting *bonobo.settings.IOFORMAT*

Default *kwargs*

Values *kwargs*, *arg0*

5.10 Examples

There are a few examples bundled with **bonobo**.

You'll find them under the `bonobo.examples` package, and you can run them directly as modules:

```
$ bonobo run -m bonobo.examples.module
```

or

```
$ python -m bonobo.examples.module
```

5.10.1 Datasets

The *bonobo.examples.datasets* package contains examples that generates datasets locally for other examples to use. As of today, we commit the content of those datasets to git, even if that may be a bad idea, so all the examples are easily runnable. Later, we'll see if we favor a "missing dependency exception" approach.

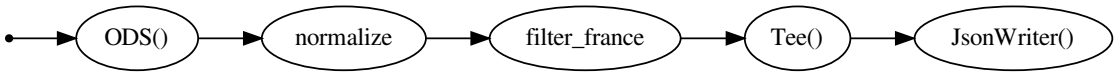
Coffeeshops

get_graph(*graph=None*, ***, *_limit=()*, *_print=()*)

Extracts a list of cafes with on euro in Paris, renames the name, address and zipcode fields, reorders the fields and formats to json and csv files.

Fablabs

Extracts a list of fablabs in the world, restricted to the ones in france, then format its both for a nice console output and a flat txt file.



get_graph(*graph=None*, ***, *_limit=()*, *_print=()*)

normalize(*row*)

5.10.2 Types

Strings

Example on how to use symple python strings to communicate between transformations.

extract()

transform(*s*)

load(*s*)

Dicts

Bags

5.10.3 Utils

Count

CHAPTER

SIX

F.A.Q.

List of questions that went up about the project, in no particuliar order.

6.1 Too long; didn't read.

Bonobo is an extract-transform-load toolkit for python 3.5+, that use regular python functions, generators and iterators as input.

By default, it uses a thread pool to execute all functions in parallel, and handle the movement of data rows in the directed graph using simple fifo queues.

It allows the user to focus on the content of the transformations, rather than worrying about optimized blocking, long operations, threads, or subprocesses.

It's lean manufacturing for data.

Note: This is NOT a «big data» tool. Neither a «data analysis» tool. We process around 5 millions database lines in around 1 hour with rdc.etl, bonobo ancestor (algorithms are the same, we still need to run a bit of benchmarks).

6.2 What versions of python does bonobo support? Why not more?

Bonobo is battle-tested against the latest python 3.5 and python 3.6. It may work well using other patch releases of those versions, but we cannot guarantee it.

The main reasons about why 3.5+:

- Creating a tool that works well under both python 2 and 3 is a lot more work.
- Python 3 is nearly 10 years old. Consider moving on.
- Python 3.5+ contains syntactic sugar that makes working with data a lot more convenient (and fun).

6.3 Can a graph contain another graph?

No, not for now. There are no tools today in bonobo to insert a graph as a subgraph.

It would be great to allow it, but there is a few design questions behind this, like what node you use as input and output of the subgraph, etc.

On another hand, if you don't consider a graph as the container but by the nodes and edges it contains, its pretty easy to add a set of nodes and edge to a subgraph, and thus simulate it. But there will be more threads, more copies of the same nodes, so it's not really an acceptable answer for big graphs. If it was possible to use a Graph as a node, then the problem would be correctly solved.

It is something to be seriously considered post 1.0 (probably way post 1.0).

6.4 How would one access contextual data from a transformation? Are there parameter injections like pytest's fixtures?

There are indeed parameter injections that work much like pytest's fixtures, and it's the way to go for transformation context.

The API may evolve a bit though, because I feel it's a bit hackish, as it is. The concept will stay the same, but we need to find a better way to apply it.

To understand how it works today, look at https://github.com/python-bonobo/bonobo/blob/master/bonobo/nodes/io/csv.py#L31 and class hierarchy.

6.5 What is a plugin? Do I need to write one?

Plugins are special classes added to an execution context, used to enhance or change the actual behavior of an execution in a generic way. You don't need to write plugins to code transformation graphs.

6.6 Is there a difference between a transformation node and a regular python function or generator?

Short answer: no.

Transformation callables are just regular callables, and there is nothing that differentiate it from regular python callables. You can even use some callables both in an imperative programming context and in a transformation graph, no problem.

Longer answer: yes, sometimes, but you should not care. The function-based transformations are plain old python callable. The class-based transformations can be plain-old-python-objects, but can also subclass Configurable which brings a lot of fancy features, like options, service injections, class factories as decorators...

6.7 Why did you include the word «marketing» in a commit message? Why is there a marketing-automation tag on the project? Isn't marketing evil?

I do use bonobo for marketing automation tasks. Also, half the job of coding something is explaining the world what you're actually doing, how to get more informations, and how to use it and that's what I call "marketing" in some commits. Even documentation is somehow marketing, because it allows a market of potential users to actually understand your product. Whether the product is open-source, a box of chips or a complex commercial software does not change a thing.

Marketing may be good or evil, and honestly, it's out of this project topic and I don't care. What I care about is that there are marketing tasks to automate, and there are some of those cases I can solve with bonobo.

6.8 Why not use <some library> instead?

I did not find the tasks I had easy to do with the libraries I tried. That may or may not apply for your cases, and that may or not include some lack of knowledge about some library from me. There is a plan to include comparisons with major libraries in this documentation, and help from experts of other libraries (python or not) would be very welcome.

See https://github.com/python-bonobo/bonobo/issues/1

Bonobo is not a replacement for pandas, nor dask, nor luigi, nor airflow… It may be a replacement for Pentaho, Talend or other data integration suites but targets people more comfortable with code as an interface.

6.9 All those references to monkeys hurt my head. Bonobos are not monkeys.

Sorry, my bad. I'll work on this point in the near future, but as an apology, we only have one word that means both «ape» and «monkey» in french, and I never realised that there was an actual difference. As one question out of two I got about the project is somehow related to primates taxonomy, I'll make a special effort as soon as I can on this topic.

Or maybe, I can use one of the comments from reddit as an answer: «Python not only has duck typing; it has the little known primate typing feature.»

See https://github.com/python-bonobo/bonobo/issues/24

6.10 Who is behind this?

Me (as an individual), and the growing number of contributors that give of their time to move the project forward.

Bonobo is not commercially endorsed, or supported. If your company wants to sponsor parts of **Bonobo** development effort, let's talk.

The code, documentation, and surrounding material is created using spare time and may lack a bit velocity. Feel free to jump in so we can go faster!

6.11 Documentation seriously lacks X, there is a problem in Y...

Yes, and sorry about that. An amazing way to make it better would be to submit a pull request about it. You can read a bit about how to contribute on page *Contributing*.

CHAPTER

SEVEN

CONTRIBUTING

There's a lot of different ways you can contribute, and not all of them includes coding. Do not think that the codeless contributions have less value, all contributions are very important.

- You can contribute to the documentation.
- You can help reproducing errors and giving more infos in the issues.
- You can open issues with problems you're facing.
- You can help creating better presentation material.
- You can talk about it in your local python user group.
- You can enhance examples.
- You can enhance tests.
- etc.

7.1 tl;dr

1. Fork the github repository

```
$ git clone https://github.com/python-bonobo/bonobo.git  # change this to use your␣
↪fork.
$ cd bonobo
$ git remote add upstream https://github.com/python-bonobo/bonobo.git
$ git fetch upstream
$ git checkout upstream/develop -b feature/my_awesome_feature
$ # code, code, code, test, doc, code, test ...
$ git commit -m '[topic] .... blaaaah ....'
$ git push origin feature/my_awesome_feature
```

2. Open pull request
3. Rince, repeat

7.2 Code-related contributions (including tests and examples)

Contributing to bonobo is usually done this way:

- Discuss ideas in the issue tracker or on Slack.
- Fork the repository.
- Think about what happens for existing userland code if your patch is applied.
- Open pull request early with your code to continue the discussion as you're writing code.
- Try to write simple tests, and a few lines of documentation.

Although we don't have a complete guide on this topic for now, the best way is to fork the github repository and send pull requests.

7.3 Tools

Issues: https://github.com/python-bonobo/bonobo/issues

Roadmap: https://www.bonobo-project.org/roadmap

Slack: https://bonobo-slack.herokuapp.com/

7.4 Guidelines

- We tend to use semantic versioning. This should be 100% true once we reach 1.0, but until then we will fail and learn. Anyway, the user effort for each BC-break is a real pain, and we want to keep that in mind.
- The 1.0 milestone has one goal: create a solid foundation we can rely on, in term of API. To reach that, we want to keep it as minimalist as possible, considering only a few userland tools as the public API.
- Said simplier, the core should stay as light as possible.
- Let's not fight over coding standards. We enforce it using yapf, and a *make format* call should reformat the whole codebase for you. We encourage you to run it before making a pull request, and it will be run before each release anyway, so we can focus on things that have value instead of details.
- Tests are important. One obvious reason is that we want to have a stable and working system, but one less obvious reason is that it forces better design, making sure responsibilities are well separated and scope of each function is clear. More often than not, the "one and only obvious way to do it" will be obvious once you write the tests.
- Documentation is important. It's the only way people can actually understand what the system do, and userless software is pointless. One book I read a long time ago said that half the energy spent building something should be devoted to explaining what and why you're doing something, and that's probably one of the best advice I read about (although, as every good piece of advice, it's more easy to repeat than to apply).

7.5 License

Bonobo is released under the apache license.

7.6 License for non lawyers

Use it, change it, hack it, brew it, eat it.

For pleasure, non-profit, profit or basically anything else, except stealing credit.

Provided without any warranty.

CHAPTER

EIGHT

FULL INDEX

PYTHON MODULE INDEX

b

INDEX

R

T