

---

# **Barabash Documentation**

***Release 0.5.0***

**Michal Nowikowski**

March 21, 2012



# CONTENTS



**WebSite** <http://barabash.99k.org>

**Download** <http://pypi.python.org/pypi/barabash/>

**Source** <https://bitbucket.org/godfryd/barabash/src>

**Mailing List** <http://groups.google.com/group/barabash/>

**Keywords** build, scripting, make alternative, scons alternative, cmake alternative

**Last updated** March 21, 2012

—  
Barabash is a build scripting framework. It takes some concepts from [GNU make](#), [CMake](#) and [SCons](#).

Fundamental assumptions:

- dependencies like in *GNU make*
- coding recipes as Bash scripts as in *GNU make* but also as Python functions as in *SCons*
- Bash recipes behave as regular scripts in contrary to *GNU make*, i.e. no more line continuations with \
- auto-clean, all explicitly generated outputs are tracked by Barabash and can be deleted on demand



# INSTALLATION

You can install Barabash either via the Python Package Index (PyPI) or from source.

To install using *pip*:

```
$ pip install -U Barabash
```

To install using *easy\_install*:

```
$ easy_install -U Barabash
```





# GETTING HELP

## 2.1 Mailing list

For any discussion about usage or development of Barabash, you are welcomed to join the [Barabash mailing list](#) .



# BUG TRACKER

If you have any suggestions, bug reports or annoyances please report them to BitBucket [issue tracker](#).



# LICENSE

MIT



# CONTRIBUTORS

Michal Nowikowski <[godfryd@gmail.com](mailto:godfryd@gmail.com)>





# CONTENTS

## 6.1 Getting Started with Barabash

### 6.1.1 Barabash Script Anatomy

Barabash script is a regular Python script that can have any name e.g. `build.py`. The script should

- import barabsh module
- define barabash operations
- invoke barabash `run()` function

Example:

```
1 from barabash import Files, MapOp, ReduceOp, run
2
3 # set of source code files definition
4 src = Files("src", ["a.c", "b.c"])
5 # compilation
6 objects = MapOp("objects", src, "%.o:%.c", "{CC} -c {in} -o {out}")
7 # linking
8 ReduceOp("program", objects, "{CC} -o {out} {in}")
9
10 run()
```

Line 1 contains Barabash imports.

Line 4 declares a set of files: *src*.

**Lines 6-8 define two Barabash operations:**

- *objects*, a map operation that compiles *.c* sources files into *.o* objects
- *program*, a reduce operation that links *.o* objects into *program* executable

Finally, line 10 starts Barabash script.

To run the script type in command line:

```
$ python build.py program
```

### 6.1.2 Operations

Operation is a base concept in Barabash. It performs an action potentially using inputs and producing outputs. Operations has a name that can be used to call the operation from command line.

Sample operation:

```
objects = MapOp("objects", src, "%o:%c", "{CC} -c {in} -o {out}")
```

where:

- “*objects*” is a name of operation
- *src* is an input for operation (dependency), it can be either `barabash.core.Files` object or some other `barabash.core.Operation` or list of them
- “*{CC} -c {in} -o {out}*” is an action

There are 3 kinds of operations in Barabash:

- map operation, defined by `barabash.core.MapOp` class,
- reduce operation, defined by `barabash.core.ReduceOp` class,
- command operation, defined by `barabash.core.Command` class.

All these classes are derived from `barabash.core.Operation` class.

Following arguments are common to all operation types.

*name* is a name of operation, it can be referenced in other `barabash.core.Operation` as dependency.

*deps* is a dependency or a list of dependencies. A dependency can be a `barabash.core.Files` or other `barabash.core.Operation`. `Files` class provides directly list of files. In case of `Operation` class it also provides list of files that are produced by this operation i.e. its outputs. This is not true only for `Command` class which does deliver any output files. It is simply required that the `Command` must be always executed.

An operation is only executed by Barabash when operation’s outputs are older then inputs, i.e. outputs must be updated. Again, this is not true for `Command` operation which is always executed.

*action* is a Python function or a Bash script that will be executed in operation. More details in [Actions](#) chapter.

## 6.1.3 Actions

Action is executed:

- when outputs of an operation are outdated comparing to its inputs in case of map and reduce operations or
- always in case of command operation.

Action can be a Python function or a Bash script as a multiline string.

Both of the action types receive input file name or names and output file name. In case of Python functions all that data is passed directly as a *env* dictionary with a predefined keys:

- *env["in"]* - a full path to input file or a list of full paths to input files,
- *env["out"]* - a full path to output file.

Example:

```
def touch_action(env):  
    os.system("touch %s" % env["out"])
```

and its usage in operation:

```
MapOp("touch", Files(["a.in"]), "%out: %in", touch_action)
```

In case of Bash script this variables can be references in script string using curly bracket notation: *{in}* and *{out}*.

Example:

```
touch {out}
```

and its usage in operation:

```
MapOp("touch", Files(["a.in"]), "%.out: %.in", "touch {out}")
```

## 6.1.4 Map Operation

Map operation is defined by `barabash.core.MapOp` class. This operation transforms input files to output files using given action one by one for each input file separately.

Map operation takes several arguments:

*MapOp(name, deps, output, action, indirect\_deps=[])*

*output* is a mapping rule like in GNU make. Example:

```
"%.o: %.c"
```

which maps .c files to .o files e.g. in compilation operation or:

```
"%.html: %.py"
```

which maps .py files to .html files e.g. in pygmentize operation.

% sign is matched to whole file path.

*indirect\_deps* is an additional set of dependent elemets (files and `barabash.core.Operation`) that is not treated as inputs, e.g. header files in C compilation.

Example:

```
# compilation
objects = MapOp("objects", src, "%.o: %.c", "{CC} -c {in} -o {out}")
```

## 6.1.5 Reduce Operation

Reduce operation is defined by `barabash.core.ReduceOp` class. This operation takes input files and transforms them in one action into one output file.

Reduce operation takes several arguments:

*ReduceOp(name, deps, output, action)*

*output* is a name of produced file, this argument can be skipped if the operation name is already a file name.

Example:

```
# linking
ReduceOp("program", objects, "{CC} -o {out} {in}")
```

## 6.1.6 Command Operation

Command operation is defined by `barabash.core.Command` class. This operation is always executed regardless of its dependencies. It does not produce any output files.

Command operation takes several arguments:

*Command(name, deps, action)*

They were already described above.

Example:

```
# list files in current directory
Command("ls", "ls -al")
```

### 6.1.7 Invoking Script

The Barabash is executed as regular Python script with expected command line arguments.

```
$ python ./build.py -h
```

output:

```
usage: build.py [-h] [-g] target
```

Barabash Build Script.

positional arguments:

target a target or help to get list of available targets

optional arguments:

-h, --help show this help message and exit

-g, --graph generate a graph image of all operations dependencies to  
barabash.png file

Pass *help* as argument to get all possible targets/operations:

```
$ python ./build.py help
```

output:

Barabash script for Barabash

Barabash Help:

upload-website

    combine-website

        build-docs

        render-website

            pygmentize-examples

        tests

build-pkg

upload-pkg

Executing specific operation – *pygmentize-examples*:

```
$ python ./build.py pygmentize-examples
```

output:

Barabash script for Barabash

BB(pygmentize-examples):

pygmentize -O style=friendly -f html -o /home/gof/barabash/www/ex3.html /home/gof/barabash/www/ex3.py

pygmentize -O style=friendly -f html -o /home/gof/barabash/www/ex1.html /home/gof/barabash/www/ex1.py

pygmentize -O style=friendly -f html -o /home/gof/barabash/www/ex2.html /home/gof/barabash/www/ex2.py

## 6.2 Advanced Scripting in Barabash

### 6.2.1 Setting environment variables

It is possible to set build parameters (called also environment variables) in four ways. They are set in following order:

- in command line
- setting variable globally in script
- using `barabash.core.set_env()` function
- in environment

#### Parameters in command line

Build parameters can be set in command line which executes Barabash script.

Example:

```
./build.py CC=/usr/bin/icc
```

#### Global variables

Build parameters can also be set as globals in Barabash script.

Example:

```
CC = "/usr/bin/icc"
objects = MapOp("objects", src, "%.o:%.c", "{CC} -c {in} -o {out}")
```

#### `barabash.core.set_env()` function

Beside setting build parameters as globals it is also possible to set them in a script using `barabash.core.set_env()` function.

Example:

```
set_env("CC", "/usr/bin/icc")
objects = MapOp("objects", src, "%.o:%.c", "{CC} -c {in} -o {out}")
```

#### Parameters from environment

The last form of setting build parameters is inheriting them from process environment. This way has the lowest precedence.

Example:

```
export CC=/usr/bin/icc
./build.py
```

or:

```
CC=/usr/bin/icc ./build.py
```

## 6.2.2 Variables in Bash actions

Bash actions are able to use environment variables. There are two types of environment variables:

- predefined variables
- user defined variables

The can be accessed in Bash action string using curly brackets:

```
"{CC} -c {in} -o {out}"
```

### Predefined variables

There are two built-in variables:

- `in`
- `out`

`in` and `out` are available in all operations beside `barabash.core.Command`.

`in` is an input to operation. In case of `barabash.core.MapOp` operation `in` is a file while in `barabash.core.ReduceOp` operation it is a list of files.

`out` is a full path to output file of an operation.

### User defined variables

There can be defined additional variables by user. It can be made in several ways what was already described in previous chapter.

### Variables' methods

It is possible to modify a variable's values in Bash action using several methods. The following methods are available:

- all from `os.path` module, e.g. `dirname` which returns the directory component of a pathname
- `droptext`, returns filename without extension (e.g. w/o `.jpg`)
- `droptext2`, drops file extensions twice (e.g. `.aa.bb`)

These methods can be applied to all variables. In the case when a values of a variable is a list then the method is executed for each element on the list.

Example:

```
ReduceOp("js.tar.gz", src_js, "tar -C {srcdir} -zcvf {out} {in.basename}")
```

Another example:

```
"""
{CC} -shared -Wl,-soname,{out.droptext} -o {out} {in}
ln -f -s {out} {out.droptext}
ln -f -s {out} {out.droptext2}
"""
```

### 6.2.3 Custom Operations

Custom behavior of operation can be implement by extending one of basic operation classes:

- `barabash.core.Operation`,
- `barabash.core.MapOp`,
- `barabash.core.ReduceOp` or
- `barabash.core.Command`.

The following methods of these classes can be overridden to get additional behavior:

- `__init__(self, ...)`
- `action(self, env)`
- `indirect_deps(self, env)`

#### `__init__(self, ...)`

Example:

```
class BuildStaticLib(core.ReduceOp):
    def __init__(self, name, sources):
        objs = Compile(name + "_compile", sources)
        super(BuildStaticLib, self).__init__(name, objs, "{AR} cru {out} {in} {RANLIB} {out}")
```

In `__init__(self, ...)` there can be added some additional instructions. In the example there is injected additional operation (*Compile*) so actually the whole operation `barabash.ops.BuildStaticLib` comprises two operations: compiling and linking static library. In `__init__(self, ...)` the initializer of super class should be always run. In this case it is an initializer of `barabash.core.ReduceOp` with appropriate arguments.

#### `action(self, env)`

Beside declaring action as function or Bash script it is also possible to directly override action in Operation subclass. A helper method `barabash.core.Operation.run_script()` is available for executing Bash script provided as string.

#### `indirect_deps(self, env)`

Another method which can be overridden is `barabash.core.Operation.indirect_deps()`. It allows providing additional indirect dependencies, e.g. `.h` header files that are included by `.c` source files.

### 6.2.4 Modules

Barabash support more complex projects by providing modules. This allows defining build sub-script for each nested component in the project.

Sample nested project:

```
prog
|--- bb-prog.py
|--- main.c
|--- liba
```

```
|      |--- main.c
|      |--- bb-liba.py
|--- libb
|      |--- bb-libb.py
|      |--- b.c
|      |--- b.h
```

*main.c* is a top level source of a *prog* program and *bb-prog.py* is top level Barabash script. *liba* is a static library which is linked into *prog*. *libb* is a dynamic library that *prog* is linking with.

*bb-prog.py*:

```
from barabash import include, BuildProgram, run

liba = include("liba/bb-liba.py")
libb = include("libb/bb-libb.py")

CFLAGS = "-DAAA=1"

BuildProgram("prog", "main.c", static_libs=liba.liba, shared_libs=libb.libb, external_libs=["m"])

run()
```

`barabash.core.include()` function is used to include modules. Then, included modules can be referenced in operations as dependencies and inputs.

*liba/bb-liba.py*:

```
from barabash import BuildStaticLib

liba = BuildStaticLib("liba.a", "main.c")
```

*libb/bb-libb.py*:

```
from barabash import BuildSharedLib

libb = BuildSharedLib("libb.so.1.0", "b.c")
```

## 6.2.5 Built-in targets

There are several built-in targets in Barabash:

- *help*
- *env*

### *help*

*help* target displays all availables targets in Barabash script:

```
$ ./build.py help
Barabash script for Barabash
Barabash Help:
upload-website
  combine-website
  build-docs
  render-website
  tests
```



```
build-pkg
upload-pkg
```

## env

*env* target display all Barabash variables that are set directly and indirectly:

```
$ ./build.py env AAAA=bar
Barabash script for Barabash
Barabash Environment:
      AAAA      bar
      AR        ar
      CC        gcc
      COLORTERM  gnome-terminal
      DESKTOP_SESSION ubuntu
      DISPLAY    :0
      GDMSESSION ubuntu
      HOME       /home/godfryd
      LANG       pl_PL.UTF-8
      LOGNAME    godfryd
      PATH       /usr/sbin:/usr/bin:/sbin:/bin
      PWD        /home/godfryd/repos/barabash2
      RANLIB     ranlib
      SHELL      /bin/bash
      SHLVL      1
      SSH_AGENT_PID 1929
      SSH_AUTH_SOCK /tmp/keyring-VUwQmc/ssh
      TERM       xterm
      USER       godfryd
      USERNAME    godfryd
      WINDOWID    62914564
      XAUTHORITY  /home/godfryd/.Xauthority
```

## 6.3 Examples

This chapter presents examples. The examples are available on BitBucket site in examples folder: <https://bitbucket.org/godfryd/barabash/src>.

### 6.3.1 Simple examples

This set of examples present various use cases of Barabash.

```
from barabash import ReduceOp, Command, MapOp, Files, set_env, run
from barabash import patsubst, define_reduce_op, define_map_op, define_ops_chain
import glob

# files
src_c = Files("src_c", ["a.c", "b.c"])
src_js = Files("src_js", ["a.js", "b.js"])

# reduce
ReduceOp("js.tar.gz", src_js, "tar -C {srcdir} -zcvf {out} {in.basename}")
```

```
# custom operation
Tar = define_reduce_op("tar -C {srcdir} -zcvf {out} {in.basename}")

Tar("js2.tar.gz", src_js)

# map + reduce
MapOp("objects", src_c, "%.o:%.c", "{CC} -c {in} -o {out}")
ReduceOp("program", objects, "{CC} -o {out} {in}")

# map + reduce: custom operations
Compile = define_map_op("%.o:%.c", "{CC} -c {in} -o {out}")
Link = define_reduce_op("{CC} -o {out} {in}")

Compile("objects2", src_c)
Link("program2", objects2)

# chaining operations
Compile = define_map_op("%.o:%.c", "{CC} -c {in} -o {out}")
Link = define_reduce_op("{CC} -o {out} {in}")
CompileAndLink = define_ops_chain(Compile, Link)

CompileAndLink("program3", src_c)

# custom command
Command("ls", [], "ls -al")

# custom command with error
Command("err", [], "rm non-existing-file")

# func as actions
def map_func(env):
    os.system("touch %s" % env['out'])
    return 0
MapOp("map_func", src_c, "%.o:%.c", map_func)

def red_func(env):
    os.system("touch %s" % env['out'])
    return 0
ReduceOp("c.c", src_c, red_func)

def cmd_func(env):
    return 4
Command("cmd_func", src_c, cmd_func)

# multiline script as action
Command("script",
    """
    a=5
    if [ $a -eq 5 ]; then
        exit $a
    fi
    """)

run()
```

### 6.3.2 Compiling nested source code

Barabash operations library provides convenient set of operations. There can be found a few operations for compiling and linking C programs and libraries (both static and shared).

This top level Barabash build script shows compiling and linking whole program. Here nested C libraries are included: one static and one shared. They are referenced in building program operation.

```
from barabash import include, BuildProgram, run

liba = include("liba/bb-liba.py")
libb = include("libb/bb-libb.py")

CFLAGS = "-DAAA=1"

BuildProgram("prog", "main.c", static_libs=liba.liba, shared_libs=libb.libb, external_libs=["m"])

run()
```

This is nested Barabash module the defines a shared C library.

```
from barabash import BuildSharedLib

libb = BuildSharedLib("libb.so.1.0", "b.c")
```

This is another nested Barabash module the defines a static C library.

```
from barabash import BuildStaticLib

liba = BuildStaticLib("liba.a", "main.c")
```

## 6.4 API Reference

### 6.4.1 Core (barabash.core)

Core of Barabash build scripting framework

```
barabash.core.run()
```

Run Barabash script.

It scans command line parameters and executes accordingly. If there is a target in command line then it is executed.

```
barabash.core.run_target(target)
```

Run specified target in Barabash script.

**Parameters** `target` – a name of target defined in Barabash script

```
barabash.core.define_map_op(output, action)
```

Define a new map operation.

**Parameters**

- **output** – a mapping definition that maps inputs to outputs, works as rules in GNU make, e.g. `%o: %c` - it maps for example `main.c` to `main.o`
- **action** – a Bash script or Python function

Example:

```
>>> Pygmentize = define_map_op("%.html:%.py", "pygmentize -O -f html -o {out} {in}")
```

`barabash.core.define_reduce_op` (*action*)

Define a new reduce operation.

**Parameters** *action* – a Bash script or Python function

Example:

```
>>> Tar = define_reduce_op("tar -C {srcdir} -zcvf {out} {in.basename}")
```

`barabash.core.define_ops_chain` (*\*actions*)

Define a chain of operations.

**Parameters** *actions* – a list of operations

Example:

```
>>> CompileAndLink = define_ops_chain(Compile, Link)
```

`barabash.core.add_dependency` (*fname, operation*)

Add dependency for given file on a given operation.

**Parameters**

- **fname** – a file name
- **operation** – an operation that file depends on

Example:

```
>>> add_dependency("www/examples.html", pygmentize_examples)
```

`barabash.core.include` (*path*)

Include submodule to Barabash script.

**Parameters** *path* – a relative or absolute path to Barabash script submodule

Example:

```
>>> lib = include("lib/lib.py")
```

`barabash.core.set_env` (*var, val*)

Set Barabash script environment variable.

**Parameters**

- **var** – a name of variable
- **val** – a value of variable

Example:

```
>>> set_env("CC", "/usr/bin/gcc")
```

`barabash.core.project` (*name*)

Define a project name.

**Parameters** *name* – a name of project

Example:

```
>>> project("Sample Project")
```

`barabash.core.patsubst` (*in\_ptrn, out\_ptrn, files*)

Substitute string in files according to pattern.

**Parameters**

- **in\_ptrn** – a pattern for input files
- **out\_ptrn** – an output pattern for output files
- **files** – a list of files

**Returns** a list of converted file names

Example:

```
>>> patsubst("%c", "%o", ["src/main.c"])
["src/main.o"]
```

**class** `barabash.core.Files` (*name*, *files=None*, *glob\_=None*, *regex=None*, *recursive=True*)

A named set of files.

**Parameters**

- **name** – a name of the set of files, it can be referenced in deps in `Operation`
- **files** – a list of files
- **glob** – a glob rule that matches a set of files, alternative to files parameter
- **regex** – a regex rule that matches a set of files, alternative to files parameter
- **recursive** – TBD, unused

Example:

```
>>> src_js = Files("src_js", ["a.js", "b.js"])
```

**class** `barabash.core.Operation` (*name*, *deps*, *op\_type*, *output*, *action=None*, *indirect\_deps=[]*)

An operation that has input dependencies and generates outputs according to action.

Operation class is a core element in Barabash that allows building a graph of dependencies.

**Parameters**

- **name** – a name of operation, it can be referenced in other `Operation` as dep
- **deps** – a dependency or a list of dependencies, a dependency can be a `Files` or other `Operation`
- **op\_type** – an operation type, it can be "map" (`MapOp`) or "reduce" (`ReduceOp`) or "cmd" (`Command`).
- **output** – a rule that defines how output is named, it depends on *op\_type*
- **action** – a python function or a Bash script that will be executed in operation
- **indirect\_deps** – an additional set of dependent elemets (files and `Operation`) that is not treated as inputs, e.g. header files in C compilation

**indirect\_deps** (*env*)

Return indirect dependencies.

Override this function in subclass to return list of additional dependencies for the operation. It can be used to bring additional deps as for example header files from C source files.

See example usage in `barabash.ops.Compile` class.

**run\_script** (*script*, *env*)

Run Bash script.

**Parameters**

- **script** – a Bash script
- **env** – variables that can be used in the script

This is helper function for running shell script. Any variables in the script are substituted with values before running. This function can be used in action function also overridden in this class.

**class** `barabash.core.MapOp` (*name, deps, output, action, indirect\_deps=[]*)

A map operation.

#### Parameters

- **name** – a name of operation, it can be referenced in other `Operation` as `dep`
- **deps** – a dependency or a list of dependencies, a dependency can be a `Files` or other `Operation`
- **output** – a mapping rule like in GNU make
- **action** – a python function or a Bash script that will be executed in operation
- **indirect\_deps** – an additional set of dependent elements (files and `Operation`) that is not treated as inputs, e.g. header files in C compilation

Example:

```
>>> objects = MapOp("objects", src_c, "%.o:%.c", "{CC} -c {in} -o {out}")
```

**class** `barabash.core.ReduceOp` (*name, deps, \*args*)

A reduce operation.

#### Parameters

- **name** – a name of operation, it can be referenced in other `Operation` as `dep`
- **deps** – a dependency or a list of dependencies, a dependency can be a `Files` or other `Operation`
- **output** – an output file name, optional
- **action** – a python function or a Bash script that will be executed in operation

Example:

```
>>> js_tar_gz = ReduceOp("js.tar.gz", src_js, "tar -C {srcdir} -zcvf {out} {in.basename}")
```

**class** `barabash.core.Command` (*name, \*args*)

A command operation.

#### Parameters

- **name** – a name of operation, it can be referenced in other `Operation` as `dep`
- **deps** – a dependency or a list of dependencies, a dependency can be a `Files` or other `Operation`
- **action** – a python function or a Bash script that will be executed in operation

Example:

```
>>> build_pkg = Command("build-pkg", "python setup.py sdist")
```

## 6.4.2 Library of Operations (barabash.ops)

Library of Barabash operations

**class** `barabash.ops.Compile` (*name, sources*)  
Compile input files.

**Parameters**

- **name** – an operation name
- **sources** – list of source files, can be either list of files or `barabash.core.Files` object

**class** `barabash.ops.BuildSharedLib` (*name, sources*)  
Build a shared library.

**Parameters**

- **name** – an operation name
- **sources** – list of source files, can be either list of files or `barabash.core.Files` object

**class** `barabash.ops.BuildStaticLib` (*name, sources*)  
Build a static library.

**Parameters**

- **name** – an operation name
- **sources** – list of source files, can be either list of files or `barabash.core.Files` object

**class** `barabash.ops.BuildProgram` (*name, sources, static\_libs=[ ], shared\_libs=[ ], external\_libs=[ ], compile\_flags=[ ], link\_flags=[ ]*)

Build a program.

**Parameters**

- **name** – an operation name
- **sources** – list of source files, can be either list of files or `barabash.core.Files` object
- **static\_libs** – list of `BuildStaticLib` operations
- **shared\_libs** – list of `BuildSharedLib` operations
- **external\_libs** – list of external libraries (e.g. ["m", "GL"] for libm.so and libGL.so)
- **compile\_flags** – TBD, not used yet
- **link\_flags** – TBD, not used yet





# INDICES AND TABLES

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



# PYTHON MODULE INDEX

## b

`barabash.core, ??`

`barabash.ops, ??`