Craftr Documentation

Release 1.1.1.dev.0

Niklas Rosenstein

July 08, 2016

Contents

Craftr is a next generation build system based on Ninja and Python that features modular and cross-platform build definitions at the flexibility of a Python script and provides access to multiple levels of build automation abstraction.

CHAPTER 1

Requirements

- Ninja
- Python 3.4 or newer

CHAPTER 2

Contents

2.1 Command-line interface

Craftr's command-line interface should feel easy, quick and efficient to use. There are only flags that alter the manifest export and build process and no subcommands.

2.1.1 Synopsis

```
usage: craftr [-h] [-V] [-v] [-m MODULE] [-b] [-e] [-c] [-d PATH] [-p PATH]
              [-D <key>[=<value>]] [-I PATH] [-N ...] [-t {standard,external}]
              [--no-rc] [--rc FILE] [--strace-depth INT] [--rts]
              [--rts-at HOST:PORT]
              [targets [targets ...]]

https://github.com/craftr-build/craftr

positional arguments:
  targets

optional arguments:
  -h, --help            show this help message and exit
  -V, --version
  -v, --verbose
  -m MODULE, --module MODULE
  -b, --skip-build
  -e, --skip-export
  -c, --clean
  -d PATH, --build-dir PATH
  -p PATH, --project-dir PATH
  -D <key>[=<value>], --define <key>[=<value>]
  -I PATH, --search-path PATH
  -N ..., --ninja-args ...
  -t {standard,external}, --buildtype {standard,external}
  --no-rc
  --rc FILE
  --strace-depth INT
  --rts
  --rts-at HOST:PORT
```

2.1.2 targets

Zero or more targets to build. Target names can be absolute or relative to the main module name (beginning with a period). Targets that are referenced from modules that haven't been imported already will be imported.

If the specified target or targets are only Python backed tasks (see *craftr.task()*), Ninja will **not** be invoked since the tasks can be executed solely on the Python side. In many cases, this is often even desired (eg. if you're using Craftr only for tasks).

2.1.3 -V, --version

Display the version of Craftr and exit immediately.

2.1.4 -v, --verbose

Add to the verbosity level of the output. This flag can be specified mutliple times. Passing the flag once will enable debug output and show module name and line number on logging from Craftr modules. Also, stracktraces are printed for *craftr.error()* uses in Craftr modules.

A verbosity level of two will enable stacktraces also for logging calls with *craftr.info()* and *craftr.warn()*.

This flag will also cause `-v` to be passed to subsequent invokations of Ninja.

2.1.5 -m, --module

Specify the main Craftr module that is initially loaded. If not specified, the Craftfile in the current working directory is loaded.

2.1.6 -b, --skip-build

Skip the build phase.

2.1.7 -e, --skip-export

Skip the export phase and, if possible, even the step of executing Craftr modules. If `-n, --no-build` is not passed, ie. building should take place, a previous invocation must have exported the Ninja build manifest before, otherwise the build can not execute.

If a manifest is present, Craftr loads the original search path (`-I`) and options (`-D`), so you don't have to specify it on the command-line again! Craftr will act like a pure wrapper for Ninja in this case.

Note that in cases where tasks are used and required for the build step, Craftr can not skip the execution phase.

Changed in v1.1.0: Inverted behaviour.

2.1.8 -c, --clean

Clean the specified targets. Pass the flag twice to clean recursively which even works without explicitly specifying a target to clean.

2.1.9 -d, --build-dir

Specify the build directory. Craftr will automatically switch to this directory before the main module is exeucted and will stay inside it until the build is completed.

2.1.10 -p, --project-dir

Similar to `-d, --build-dir`, but this option will cause Craftr to use the current working directory as build directory and instead load the main module from the specified project directory.

2.1.11 -D, --define

Format: `-D key[=value]`

Set an option, optionally with a specific string value. This option is set in the environment variables of the Craftr process and inherit by Ninja. The `key` may be anything, but if it begins with a period, it will be automatically prefixed with the main module identifier.

As an example, say the Craftfile in your working directory has the identifier `my_module`. Using `-D.debug=yes` will set the environment variable `my_module.debug` to the string `'yes'`.

If you leave out the value part, the option is set to the string value `'true'`. If you keep the assignment operator without value, the option will be *unset*.

2.1.12 -I, --search-path

Add an additional search path for Craftr modules.

2.1.13 -N, --ninja-args

Consumes all arguments after it and passes it to the Ninja command in the build step.

2.1.14 -t, --buildtype {standard, external}

Switch between standard or externally controlled build. Choosing the `external` option will cause target generator functions to consider environment variables like `CFLAGS`, `CPPFLAGS`, `LDFLAGS` and `LDLIBS` or whatever else is applicable to the target generator you're using.

Note: The consideration of these environment variables is completely dependent on the implementation of the target generator.

See also:

The selected buildtype can be read from the *craftr.Session.buildtype* attribute.

2.1.15 --no-rc

Don't run `craftrc.py` files

2.1.16 --rc

Specify a file that will be executed before anything else. It will be executed the same way `craftrc.py` files are. Can be combined with `--no-rc` to exclusively run the specified file.

2.1.17 --strace-depth

Specify the depth of the stacktrace when it is printed. This is only for stacktraces printed with the *Logging*. The default value is 5. Also note that frames of builtin modules are hidden from this stacktrace.

2.1.18 --rts

Keep alive the Craftr runtime server until you quit it with CTRL+C.

2.1.19 --rts-at

Specify the `HOST:PORT` for the Craftr runtime server instead of picking loopback and a random port.

2.2 Craftr Standard Library

2.2.1 Standard Library Modules

craftr.ext.archive

Classes

class craftr.ext.archive.**Archive**(*name=None*, *base_dir=None*, *prefix=None*, *format='zip'*)

Helper class to build and a list of files for an archive and then create that archive from that list. If no *name* is specified, it is derived from the *prefix*. The *format* must be `'zip'` for now.

add(*name*, *rel_dir=None*, *arc_name=None*, *parts=None*)

Add a file, directory or *Target* to the archive file list. If *parts* is specified, it must be a number which specifies how many parts of the arc name are kept from the right.

Note: *name* can be a filename, the path to a directory, a glob pattern or list. Note that a directory will be globbed for its contents and will then be added recursively. A glob pattern that yields a directory path will add that directory.

---

exclude(*filter*)

Remove all files in the Archive's file list that match the specified *filter*. The filter can be a string, in which case it is applied with `fnmatch()` or a function which accepts a single argument (the filename).

rename(*old_arcname*, *new_arcname*)

Rename the *old_arcname* to *new_arcname*. This will take folders into account.

save()

Save the archive.

craftr.ext.cmake

CMake-style file configuration.

```
from craftr import path
from craftr.ext import cmake

cvconfig = cmake.configure_file(
  input = path.local('cmake/templates/cvconfig.h.in'),
  environ = {
    'BUILD_SHARED_LIBS': True,
    'CUDA_ARCH_BIN': '...',
    # ...
  }
)

info('cvconfig.h created in', cvconfig.include)
```

Functions

craftr.ext.cmake.**configure_file**(*input*, *output=None*, *environ={}*, *inherit_environ=True*)

Renders the CMake configuration file using the specified environment and optionally the process' environment.

If the *output* parameter is omitted, an output filename in a special `include/` directory will be generated from the *input* filename. The `.in` suffix from *input* will be removed if it exists.

Parameters

- **input** – Absolute path to the CMake config file.
- **output** – Name of the output file. Will be automatically generated if omitted.
- **environ** – A dictionary containing the variables for rendering the CMake configuration file. Non-existing variables are considered undefined.
- **inherit_environ** – If True, the environment variables of the Craftr process are additionally taken into account.

Returns A *ConfigResult* object.

Classes

class craftr.ext.cmake.**ConfigResult**(**args*, ***kwargs*)

craftr.ext.compiler

This module provides common utility functions that are used by compiler interface implementations, for example to convert source filenames to object filenames using *gen_objects()*.

Functions

craftr.ext.compiler.**detect_compiler**(*program*, *language*)

Detects the compiler interface based on the specified *program* assuming it is used for the specified *language*. Returns the detected compiler or raises *ToolDetectionError*. Supports all available compiler toolset implementations.

craftr.ext.compiler.**gen_output_dir**(*output_dir*)
Given an output directory that is a relative path, it will be prefixed with the current modules' project name. An absolute path is left unchanged. If None is given, the current working directory is returned.

craftr.ext.compiler.**gen_output**(*output*, *output_dir=''*, *suffix=None*)

craftr.ext.compiler.**gen_objects**(*sources*, *output_dir='obj'*, *suffix=None*)

craftr.ext.compiler.**remove_flags**(*command*, *remove_flags*, *builder=None*)
Helper function to remove flags from a command.

Parameters

- **command** – A list of command-line arguments.
- **remove_flags** – An iterable of flags to remove.
- **builder** – Optionally, a *craftr.TargetBuilder* that will be used for logging.

Returns The "command" list, but it is also directly altered.

Exceptions

class craftr.ext.compiler.**ToolDetectionError**
This exception is raised if a command-line tool could not be successfully be detected.

Submodules

craftr.ext.compiler._base Provides a convenient base class for Craftr compilers.

craftr.ext.compiler.base Provides a convenient base class for Craftr compilers.

class craftr.ext.compiler.base.**BaseCompiler**(***kwargs*)
This is a convenient base class for implementing compilers.

Params kwargs Arbitrary keyword arguments from which a `Framework` will be created and assigned to the *settings* member

```
from craftr.ext.compiler.base import BaseCompiler
from craftr.ext.compiler import gen_output

class SimpleGCC(BaseCompiler):
  def compile(self, sources, frameworks, **kwargs):
    builder = self.builder(sources, frameworks, kwargs)
    include = builder.merge('include')
    defines = builder.merge('defines')

    outputs = gen_output(builder.input, suffix='.obj')
    command = ['gcc', '-c', '$in', '-c', '-o', '$out']
    command += ['-I' + x for x in include]
    command += ['-D' + x for x in defines]
    return builder.create_target(command, outputs, foreach=True)
```

In the above example, the `TargetBuilder` returned by *builder()* has the following framework option resolution order (first is first):

1. The `**kwargs` passed to `compile()`
2. The `Framework` objects in `frameworks`

3.The *settings* framework of SimpleGCC

4.If the sources list contained an Target s, the Framework s of these targets will be considered

settings
A Framework that will be included in the TargetBuilder returned by the *builder()* method.

builder(*inputs*, *frameworks*, *kwargs*, ***_add_kwargs*)
Create a TargetBuilder that includes the *settings* Framework of this *BaseCompiler*.

fork(***kwargs*)
Create a fork of the compiler that overrides/add parameters in the *settings* with the specified **kwargs.

register_hook(*call*, *handler*)
Registers a handler for the method call that will be invoked when a TargetBuilder was created. It will allow the "handler" to set up default and additional settings.

craftr.ext.compiler.csc

class craftr.ext.compiler.csc.**CSCompiler**(*program='csc'*)
Class for compiling C-Sharp programs using Microsoft CSC.

compile(*filename*, *sources*, *target='exe'*, *defines=()*, *optimize=True*, *warn=None*, *warnaserror=False*, *appconfig=None*, *baseaddress=None*, *checked=False*, *debug=False*, *main=None*, *platform=None*, *unsafe=False*, *win32icon=None*, *win32manifest=None*, *win32res=None*, *additional_flags=()*)

craftr.ext.compiler.cython Interface for compiling Cython source code. See also Using Craftr for Cython projects.

class craftr.ext.compiler.cython.**CythonCompiler**(*program=None*, *detect=True*, ***kwargs*)
Compiler interface for Cython. Note that this class does not provide functionality to actually compile the C/C++ source files generated by Cython.

A small example:

```
from craftr import path, options
from craftr.ext.compiler.cython import cythonc

c_files = cythonc.compile(
  py_sources = path.glob('mymodule/**/*.pyx'),
  python_version = int(options.get('python_version', 3)),
  fast_fail = True,
  cpp = True,
)
```

compile(*py_sources*, *outputs=None*, *frameworks=()*, *target_name=None*, ***kwargs*)
Compile the specified *py_sources* files to C or C++ source files.

Parameters

- **py_sources** – A list of .pyx or .py files.
- **outputs** – Override the output filenames. If omitted, default output filenames are generated.
- **frameworks** – List of additional frameworks.
- **target_name** – Alternative target name.
- **include** – Additional include directories for Cython.

- **fast_fail** – True to enable the `--fast-fail` flag.
- **cpp** – True to translate to C++ source files.
- **embed** – Pass `--embed` to Cython. Note that if multiple files are specfied in "py_sources", all of them will have a `int main()` function.
- **additional_flags** – List of additional flags for the Cython command.
- **python_version** – The Python version to build for (2 or 3). Defaults to 3.

Produces the following meta variables in the returned target:

•cython_outdir – The common output directory of the Cython source files

compile_project(*main=None*, *sources=[]*, *python_bin='python'*, *cc=None*, *ld=None*, *defines=()*, ***kwargs*)

Compile a set of Cython source files into dynamic libraries for the Python version specified with "python_bin".

Parameters

- **main** – Optional filename of a `.pyx` file that will be compiled with the `--embed` option and compiled to an executable file.
- **sources** – A list of the *.pyx* source files.
- **python_bin** – The name of the Python executable to compile for.
- **cc** – Alternative C/C++ compiler implementation. Defaults to `platform.cc`
- **ld** – Alternative linker implementation. Defaults to `platform.ld`
- **defines** – Additional defines for the compiler invokation.

Returns A `ProjectResult` object

name = 'Cython'

class craftr.ext.compiler.cython.**PythonInfo**(*pybin*)

Container class for meta information of an installed Python version. The information is read from the *craftr.ext.python* module.

fw

The framework retrieved with `get_python_framework()`

conf

The Python version's setuptools configuration retrieved with `get_python_config_vars`.

major_version

Returns the major version number of the Python installation.

craftr.ext.compiler.cython.**cythonc = <craftr.ext.compiler.cython.CythonCompiler object>**

An instance of the *CythonCompiler* created with the default arguments.

craftr.ext.compiler.flex

class craftr.ext.compiler.flex.**FlexCompiler**(*program='flex'*)

Interface for the lex compiler.

compile(*sources*, *output=None*, *debug=False*, *fast=False*, *case_insensitive=False*, *max_compatibility=False*, *performance_report=False*, *no_warn=False*, *interactive=False*, *bits=None*, *cpp=False*, *compress=None*, *prefix=None*)

craftr.ext.compiler.gcc

craftr.ext.compiler.gcc.**detect**(*program*)

Assuming *program* points to GCC or GCC++, this function determines meta information about it. The returned dictionary contains the following keys:

- version
- version_str
- name
- target
- thread_model
- cpp_stdlib (only present for GCC++)

Raises

- **OSError** – If *program* can not be executed (eg. if it does not exist).
- **ToolDetectionError** – If *program* is not GCC or GCC++.

class craftr.ext.compiler.gcc.**GccCompiler**(*program*, *language='c'*, *desc=None*, ***kwargs*)

Interface for the GCC compiler.

Note: Currently inherits the LLVM implementation. Will eventually get its own implementatio in the future, but not as long as the LLVM version works well for GCC, too.

---

name = 'GCC (Craftr-LLVM-Backend)'

craftr.ext.compiler.java

craftr.ext.compiler.java.**get_class_files**(*sources*, *source_dir*, *output_dir*)

class craftr.ext.compiler.java.**JavaCompiler**(*javac='javac'*, *jar='jar'*)

Class for compiling Java source files using the java compiler.

compile(*source_dir*, *sources=None*, *debug=False*, *warn=True*, *classpath=()*, *additional_flags=()*)

get_version()

Returns a tuple of *(name, version)*.

make_jar(*filename*, *classes*, *entry_point=None*)

craftr.ext.compiler.llvm

craftr.ext.compiler.llvm.**detect**(*program*)

Assuming *program* points to Clang or Clang++, this function determines meta information about it. The returned dictionary contains the following keys:

Parameters

- **version** –
- **version_str** –
- **name** –
- **target** –
- **thread_model** –
- **cpp_stdlib** – (only present for C++ compilers)

Raises

- **OSError** – If *program* can not be executed (eg. if it does not exist).
- **ToolDetectionError** – If *program* is not Clang or Clang++.

class craftr.ext.compiler.llvm.**LlvmCompiler**(*program*, *language*, *desc=None*, ***kwargs*)

Interface for the LLVM compiler.

compile(*sources*, *frameworks=()*, *target_name=None*, ***kwargs*)

Parameters

- **sources** – A list of input source files.
- **frameworks** – List of `Framework` objects.
- **target_name** – Override target name.

Supported framework options:

Parameters

- **include** – Additional include directories.
- **defines** – Preprocessor definitions.
- **forced_include** – Force includes for every compilation unit.
- **exceptions** – Allows you to disable exceptions.
- **language** – Override compilation language. Choices are `'c'`, `'cpp'`, `'asm'`
- **debug** – True ot disable optimizations and enable debugging symbols.
- **std** – Set the C/C++ standard (`--std` argument)
- **pedantic** – Enable the `--pedantic` flag
- **pic** – Enable position independent code.
- **warn** – Warning level. Choices are `'all'`, `'none'` and `None` (latter is different in that it adds no warning related compiler flag at all).
- **optimize** – Optimization level. Choices are `'debug'`, `'speed'`, `'size'`, `'none'` and `None`
- **autodeps** – True if automatic dependencies should be enabled (for recompiles when only headers change). Default is True.
- **description** – Target description (shown during Ninja build).
- **osx_fwpath** – Additional search path for OSX frameworks.
- **osx_frameworks** – OSX frameworks to take into account.
- **program** – Override the compiler command.
- **additional_flags** – Additional flags for the compiler command-
- **gcc_additional_flags** – Additional flags (GCC only).
- **gcc_compile_additional_flags** – Additional flags (GCC only).
- **gcc_remove_flags** – Flags to remove (GCC only).
- **gcc_compile_remove_flags** – Flags to remove (GCC only).
- **llvm_additional_flags** – Additional flags (LLVM only).
- **llvm_compile_additional_flags** – Additional flags (LLVM only).

- **llvm_remove_flags** – Flags to remove (LLVM only).
- **llvm_compile_remove_flags** – Flags to remove (LLVM only).

link(*output*, *inputs*, *frameworks=()*, *target_name=None*, ***kwargs*)

Parameters

- **output** – The name of the output file. The platform-dependent appropriate suffix is automatically appended unless *keep_suffix* is True.
- **inputs** – A list of input files/targets.
- **frameworks** – List of additional `Framework` objects. Note that the frameworks of `Target` objects listed in *inputs* are taken into account automatically.
- **target_name** – Override target name.

Supported framework options:

Parameters

- **output_type** – The output type. Can be `'bin'` or `'dll'`
- **keep_suffix** – Do not replace the suffix of the specified *output* files.
- **debug** – True to enable debug symbols and disable optimization.
- **libs** – Additional library names to link with.
- **gcc_libs** – Additional library names to link with (GCC only).
- **llvm_libs** – Additional library names to link with (LLVM only).
- **linker_args** – Additional linker aguments.
- **gcc_linker_args** – Additional linker aguments (GCC only).
- **llvm_linker_args** – Additional linker aguments (LLVM only).
- **linker_script** – Linker script input file.
- **libpath** – Additional search directory to search for libraries.
- **external_libs** – Absolute paths of additional libraries to link with.
- **osx_fwpath** – Additional search path for frameworks (OSX only).
- **osx_frameworks** – Frameworks to link with (OSX only).
- **description** – Target description (displayed during Ninja build).
- **program** – Override the linker program to incoke.
- **additional_flags** – Additional flags for the linker.
- **gcc_additional_flags** – Additional flags for the linker (GCC only).
- **gcc_link_additional_flags** – Additional flags for the linker (GCC only).
- **gcc_remove_flags** – Flags to remove (GCC only).
- **gcc_link_remove_flags** – Flags to remove (GCC only).
- **llvm_additional_flags** – Additional flags for the linker (LLVM only).
- **llvm_link_additional_flags** – Additional flags for the linker (LLVM only).
- **llvm_remove_flags** – Flags to remove (LLVM only).
- **llvm_link_remove_flags** – Flags to remove (LLVM only).

`Target.meta` variables:

Parameters

- **link_output** – The output filename of the link operation.
- **link_target** – The filename of the target that can be passed into the linker. This is required because on Windows this needs to be a different value than `link_output`. Only valid with `output_type='dll'`.

name = 'LLVM'

craftr.ext.compiler.msvc

`craftr.ext.compiler.msvc.`**detect**(*program*)

Detects the version of the MSVC compiler from the specified *program* name and returns a dictionary with information that can be passed to the constructor of *MsvcCompiler* or raises *ToolDetectionError*.

This function also supports detecting the Clang-CL compiler.

Parameters **program** – The name of the program to execute and check.

Returns

`dict` of

- name (either `'msvc'` or `'clang-cl'`)
- version
- version_str
- target
- thread_model
- msvc_deps_prefix

Raises

- **OSError** – If *program* can not be executed (eg. if it does not exist).
- **ToolDetectionError** – If *program* is not GCC or GCC++.

`craftr.ext.compiler.msvc.`**get_vs_install_dir**(*versions=None*, *prefer_newest=True*)

Returns the path to the newest installed version of Visual Studio. This is determined by reading the environment variables `VS***COMNTOOLS`.

If "versions" is specified, it must be a list of three-digit version numbers like `100` for Visual Studio 2010, `110` for 2012, `120` for 2013, `140` for 2015, etc.

Parameters

- **versions** – Optionally, a list of acceptable Visual Studio version numbers that will be considered. If specified, the first detected installation will be used.
- **prefer_newest** – True if the newest version should be preferred.

Returns `str` of the main installation directory.

Raises **ToolDetectionError** – If no Visual Studio insallation could be found.

Note: The option `VSVERSIONS` can be used to override the "versions" parameter if no explicit value is specified.

---

craftr.ext.compiler.msvc.**get_vs_environment**(*install_dir*, *arch=None*)

Given an installation directory returned by *get_vs_install_dir()*, returns the environment that is created from running the Visual Studio vars batch file.

Parameters

- **install_dir** – The installation directory.
- **arch** – The architecture name. If no value is specified, an architecture matching the current host operating system is selected.

Note: The option VSARCH can be used to specify the default value for "arch" if no explicit value is specified.

---

class craftr.ext.compiler.msvc.**MsvcCompiler**(*program='cl'*, *language='c'*, *desc=None*, ***kwargs*)

Interface for the MSVC compiler.

Parameters

- **program** – The name of the MSVC compiler program. If not specified, cl will be tested, otherwise *get_vs_install_dir()* will be used.
- **language** – The language name to compile for. Must be c, c++ or asm.
- **desc** – The description returned by *detect()*. If not specified, *detect()* will be called by the constructor.
- **kwargs** – Additional arguments that will be taken into account as a Framework to *compile()*.

compile(*sources*, *frameworks=()*, *target_name=None*, *meta=None*, ***kwargs*)

Supported options:

•language

•include (/I) [list of str]

•defines (/D) [list of str]

•forced_include (/FI) [list of str]

•debug (/Od /Zi /RTC1 /FC /Fd /FS) [True, False]

•warn (/W4, /w) ['all', 'none', None]

•optimize (/Od, /O1, /O2, /Os) ['speed', 'size', 'debug', 'none', None]

•exceptions (/EHsc) [True, False, None]

•autodeps (/showIncludes)

•description

•msvc_runtime_library (/MT, /MTd, /MD, /MDd) ['static', 'dynamic', None]

•msvc_disable_warnings (/wd) [list of int/str]

•program

•additional_flags

•msvc_additional_flags

•msvc_compile_additional_flags

•msvc_remove_flags

- msvc_compile_remove_flags
- msvc_use_default_defines

Unsupported options supported by other compilers:

- std
- pedantic
- pic
- osx_fwpath
- osx_frameworks

Target meta variables: *none*

name = 'msvc'

class `craftr.ext.compiler.msvc.`**`MsvcLinker`**(*program='link'*, *desc=None*, ***kwargs*)

Interface for the MSVC linker.

`link`(*output*, *inputs*, *frameworks=()*, *target_name=None*, *meta=None*, ***kwargs*)

Supported options:

- output_type
- keep_suffix
- libpath
- libs
- msvc_libs
- win32_libs
- win64_libs
- external_libs
- msvc_external_libs
- debug
- description
- program
- additional_flags
- msvc_additional_flags
- msvc_link_additional_flags
- msvc_remove_flags
- msvc_link_remove_flags

Target meta variables:

- link_output – The output filename of the link operation.
- link_target – The filename that can be specified to the linker. This is necessary because on Windows you pass in a separately created `.lib` file instead of the `.dll` output file.

name = 'msvc:link'

class craftr.ext.compiler.msvc.**MsvcAr**(*program='lib'*, ***kwargs*)
Interface for the MSVC lib tool.

name = 'msvc:lib'

staticlib(*output*, *inputs*, *export=()*, *frameworks=()*, *target_name=None*, *meta=None*, ***kwargs*)
Supported options:

- program
- additional_flags
- msvc_additional_flags
- msvc_staticlib_additional_flags
- description

Target meta variables:

- staticlib_output – The output filename of the library operation.

class craftr.ext.compiler.msvc.**MsvcSuite**(*vsversions=None*, *vsarch=None*)
Represents an MSVC installation and its meta information.

craftr.ext.compiler.nvcc

class craftr.ext.compiler.nvcc.**NvccCompiler**
Interface for the NVIDIA CUDA compiler. Uses the environment variable `CUDA_PATH` to determine the CUDA toolkit location.

Important: This has been tested on Windows only, yet.

---

compile(*sources*, *machine=64*, *static=True*)

get_opencl_context(*arch=64*)

get_opencl_framework(*arch=64*)

craftr.ext.compiler.protoc

craftr.ext.compiler.protoc.**get_proto_meta**(*filename*)
Extracts the package declaration and various meta information from the specified .proto file.

class craftr.ext.compiler.protoc.**ProtoCompiler**(*program='protoc'*)
Interface for the Google Protocol Buffers Compiler.

compile(*sources*, *proto_path=()*, *cpp_out=None*, *java_out=None*, *python_out=None*)

craftr.ext.compiler.yacc

class craftr.ext.compiler.yacc.**YaccCompiler**(*program='yacc'*)
Interface for yacc.

compile(*infile*, *output=None*, *prefix=None*, *backtracing=False*, *write_defs=False*, *write_interface=False*, *write_graphic=False*, *symbol_prefix=None*, *reentrant=False*, *debug=False*)

craftr.ext.git

A very small interface for querying information about a Git repository.

Examples

Display a note in console if build is started with unversioned changes in the Git repository.

```
git = load_module('git').Git(project_dir)
info('Current Version:', git.describe())
if git.status(exclude='??'):
  info('Unversioned changes present.')
```

Export a `GIT_VERSION.h` header file into the build directory (not to mess with your source tree!)

```
from craftr import *
from craftr.ext import git

def write_gitversion():
  filename = path.buildlocal('include/GIT_VERSION.h')
  dirname = path.dirname(filename)
  if session.export:
    path.makedirs(dirname)
    description = git.Git(project_dir).describe()
    with open(filename, 'w') as fp:
      fp.write('#pragma once\n#define GIT_VERSION "{}"\n'.format(description))
  return dirname

gitversion_dir = write_gitversion()  # Add this to your includes
```

Classes

class craftr.ext.git.**Git**(*git_dir*)

branch()

branches()

describe(*mode='tags'*, *all=False*, *fallback=True*)

status(*include=None*, *exclude=None*)

craftr.ext.platform

This module represents the current platform that Craftr is running on by importing the correct implementation based on `sys.platform`. Be sure to check out the *Platform Interface* documentation.

Platform C/C++ Toolset

craftr.ext.platform.**asm**
: The Assembler retrieved with *platform.get_tool()*

craftr.ext.platform.**cc**
: The C compiler retrieved with *platform.get_tool()*

craftr.ext.platform.**cxx**
: The C++ compiler retrieved with *platform.get_tool()*

craftr.ext.platform.**ld**
: The linker retrieved with *platform.get_tool()*

craftr.ext.platform.**ar**
The archiver retrieved with *platform.get_tool()*

Constants

craftr.ext.platform.**WIN32 = 'win'**
Windows platform name

craftr.ext.platform.**DARWIN = 'mac'**
Mac OS platform name

craftr.ext.platform.**LINUX = 'linux'**
Linux platform name

craftr.ext.platform.**CYGWIN = 'cygwin'**
Cygwin platform name

Submodules

craftr.ext.platform.cygwin

craftr.ext.platform.darwin

craftr.ext.platform.linux

craftr.ext.platform.win32

craftr.ext.python

This Craftr extension module provides information about Python installations that are required for compiling C-extensions. Use the *get_python_framework()* function to extract all the information from a Python installation using its `distutils` module.

craftr.ext.python.**get_python_config_vars**(*python_bin*)
Given the name or path to a Python executable, this function returns the dictionary that would be returned by `distutils.sysconfig.get_config_vars()`.

craftr.ext.python.**get_python_framework**(*python_bin*)
Uses *get_python_config_vars()* to read the configuration values and returns a `Framework` from that data that exposes the following options:

Variables

- **include** – List of include paths (derived from `INCLUDEPY`)
- **libpath** – List of library paths (derived from `LIBDIR`)

craftr.ext.rules

craftr.ext.rules.**alias**(**targets*, *target_name=None*)
Create an alias target that causes all specified "targets" to be built.

Parameters

- **targets** – The targets to create an alias for. You may pass None for an element, in which case it is ignored.
- **target_name** – Alternative target name.

craftr.ext.rules.**run**(*commands*, *args=()*, *inputs=()*, *outputs=None*, *cwd=None*, *pool=None*, *description=None*, *target_name=None*, *multiple=False*)

This function creates a `Target` that runs a custom command. The function is three different modes based on the first parameter.

1. If *commands* is a `Target`, that target must list exactly one file in its outputs and that file is assumed to be a binary and will be executed by the target created by this function. The *args* parameter may be a list of additional arguments for the program.
2. If *commands* is a list, it is handled as a list of commands, never as a single command. Thus a string in the list represents a complete command, as does a list of strings (representing the command as its individual arguments).
3. If *commands* is a string, it will be treated as a single command.

If multiple commands need to be invoked, `TargetBuilder.write_multicommand_file` is used to create a script to invoke multiple commands.

__Examples__

```
main = ld.link(
  output = 'main',
  inputs = objects,
)
run = rules.run(main, args = [path.local('testfile.dat')])
```

```
run = rules.run([
  'command1 args11 args12 args13',
  ['command2', 'args21', 'args22', 'args23'],
], cwd = path.local('test'), multiple=True)
```

Parameters

- **commands** – A `Target`, string or list of strings/command lists.
- **args** – Additional program arguments when a `Target` is specified for *commands*.
- **inputs** – A list of input files for the command. These can be referenced using the Ninja variable `%in` in the command(s).
- **outputs** – A list of outputs generated by the command. These can be referenced using the Ninja variable `%out` in the command(s).
- **cwd** – An optional working directory to switch to when executing the command(s). If None is passed, the build directory is used.
- **pool** – Override the default pool that the command is executed in. If a `Target` is passed for *commands*, this will default to `console`.
- **description** – Optional target description displayed when building with Ninja.
- **multiple** – True if *commands* is a list of commands. This will cause a shell/batch script to be created and invoked by Ninja.
- **target_name** – An optional override for the return target's name.

Returns A `Target`.

craftr.ext.rules.**render_template**(*template*, *output*, *context*, *env=None*, *target_name=None*)
Creates a `task()` that renders the file *template* using Jinja2 with the specified *context* to the *output* file.

```
# craftr_module(my_project)

import jinja2
from craftr import path
from craftr.ext import rules

# We can use the render_template() task factory to render
# a Jinja2 template that outputs a linker script.
ld_script = rules.render_template(
  template = path.local('my_project.ld.jinja2'),
  output = 'test.html',
  env = jinja2.Environment(
    variable_start_string = '{$',
    variable_end_string = '$}',
  ),
  context = dict(
    # Context variables here
  )
)
```

Parameters

- **template** – Filename of a Jinja template.
- **output** – Output filename.
- **context** – Context dictionary.
- **env** – A `jinja2.Environment` object.
- **target_name** – Optional target name. Automatically deduced from the assigned variable if omitted.

craftr.ext.unix

craftr.ext.unix.**pkg_config**(**flags*)
Calls *pkg-config* with the specified flags and returns a list of the returned flags.

class craftr.ext.unix.**Ar**(*program='ar'*, ***kwargs*)
Interface for the Unix *ar* archiver.

name = 'Unix AR'

staticlib(*output*, *inputs*, *target_name=None*, *meta=None*, ***kwargs*)
Supported options:

•program

•ar_additional_flags – A string of additional flags (not a list!)

Target meta variables:

•staticlib_output – The output filename of the library operation.

class craftr.ext.unix.**Ld**(*program='ld'*, ***kwargs*)
Interface for the Unix *ld* command.

link(*output*, *inputs*, *frameworks=()*, *target_name=None*, *meta=None*, ***kwargs*)
Supported options:

- program
- linker_script

Target meta variables:

- link_output – The output filename of the link operation.

name = 'Unix LD'

class craftr.ext.unix.**Objcopy**(*program='objcopy'*, *detect=True*, ***kwargs*)

Interface for the *objcopy* tool.

name = 'Unix Objcopy'

objcopy(*output_format*, *inputs*, *outputs=None*, *target_name=None*, *output_dir=''*, *meta=None*, ***kwargs*)

Performs an objcopy task with an output file (no append!) given the specified *inputs* generating *outputs* with the specified *output_format*. If *outputs* is omitted, it will be automatically generated from *inputs*.

Supported options:

- program
- output_suffix
- input_format
- binary_architecture
- description

Target meta variables: *none*

2.2.2 General Properties

Compiler implementations should consider the `'debug'` option when handling the build parameters. More specifically, given a target uses a `TargetBuilder`, it is usually good practice to read the debug option like this:

```
debug = builder.get('debug', options.get_bool('debug'))
```

2.2.3 Platform Interface

All `platform.xxx` modules implement this interface.

`platform.`**`name`**

A string identifier of the platform. Currently implemented values are

- `'win'`
- `'cygwin'`
- `'linux''`
- `'darwin'`

`platform.`**`standard`**

A string identifier of the platform standard. Currently implemented values are

- `'nt'`
- `'posix'`

`platform.`**`obj`**`(`*x*`)`

Given a filename or list of filenames, replaces all suffixes with the appropriate suffix for compiled object files for the platform.

`platform.`**`bin`**`(`*x*`)`

Given a filename or list of filenames, replaces all suffixes with the appropriate suffix for binary executable files for the platform.

`platform.`**`dll`**`(`*x*`)`

Given a filename or list of filenames, replaces all suffixes with the appropriate suffix for shared library files for the platform.

`platform.`**`lib`**`(`*x*`)`

Given a filename or list of filenames, replaces all suffixes with the appropriate suffix for static library files for the platform.

`platform.`**`get_tool`**`(`*name*`)`

Given the name of a tool, returns an object that implements the respective tools interface. The returned object may already consider environment variables like `CC` and `CXX`. Possible values for *name* are

Name	Description
`'c'`	C Compiler (see *C/C++ Compiler Interface*)
`'c++'`	C++ Compiler (see *C/C++ Compiler Interface*)
`'asm'`	ASM Compiler (see *C/C++ Compiler Interface*)
`'ld'`	Linker (usually the same as C compiler on Linux/Mac OS) (see *Linker Interface*)
`'ar'`	Static libary generator (archiver) (see *Archiver Interface*)

2.2.4 C/C++ Compiler Interface

`compiler.`**`compile`**`(`*sources*, *frameworks=()*, *target_name=None*, ***kwargs*`)`

Target.meta output variables:	
None	

Known Implementations

- *craftr.ext.compiler.msvc.MsvcCompiler.compile()*
- *craftr.ext.compiler.llvm.LlvmCompiler.compile()*

2.2.5 Linker Interface

`linker.`**`link`**`(`*output*, *inputs*, *output_type='bin'*, *frameworks=()*, *target_name=None*, ***kwargs*`)`

Target.meta output variables:	
`'link_output'` `'link_target'`	Absolute output filename Linker target filename (1)

(1) This is required because on Windows you can not passed the actuall DLL filename to the linker but you must pass to it the also generated `.lib` file which is what this `'link_target'` value is pointing to. Other implementations like GCC/LLVM just fill in the same filename as in `'link_output'`

Known Implementations

- *craftr.ext.compiler.msvc.MsvcLinker.link()*
- *craftr.ext.compiler.llvm.LlvmCompiler.link()*

2.2.6 Archiver Interface

archiver.**staticlib**(*output*, *inputs*, *target_name=None*, ***kwargs*)

Target.meta output variables:	
'staticlib_output'	Absolute output filename

Known Implementations

- *craftr.ext.compiler.msvc.MsvcAr.staticlib()*
- *craftr.ext.unix.Ar.staticlib()*

2.3 Extension Modules

Craftr comes with a set of builtin modules that contain useful functionality to quickly write powerful Craftfiles. Most of the modules contain compiler classes which in turn expose rule functions (ie. functions with a high level interface that produce low-level targets). For more information on the standard library, see Craftr Standard Library.

2.3.1 A primer on Craftr modules

While Craftr modules can be imported from a Craftfile like any other Python module, they are sligthly different in the file structure to make them easier to use for common build scenarios. There are two ways to create a Craftr module:

1. A `Craftfile.py` file with a `#craftr_module(<module_name>)` declaration at the top of the file
2. A `craftr.ext.<module_name>.py` file

While 2) is used more commonly for pure extension modules (eg. the whole standard library of Craftr is built of those files), 1) is preferred for the main build module of a project. There is no technical difference between these two types of files though.

2.3.2 Importing Craftr Modules

The *craftr.Session* object manages a list of search paths for Craftr modules. It is important to note that the Craftr modules in this search path must **not** be directly inside the listed directories, but they are additionally searched for one level deeper in the folder structure.

Consider the following project structure:

```
my_project/
  Craftfile.py
  src/
  vendor/
    qt5/
      craftr.ext.qt5.py
```

In order to be able to import the Qt5 module, you only need to add the `vendor/` directory to the search path! This is a design decision that was made for plain convenience.

```
#craftr_module(my_project)
from craftr import *
session.path.append(path.local('vendor'))
from craftr.ext import qt5
```

2.4 Tutorials

2.4.1 Using Craftr for C++ projects (TODO)

Todo

Nice tutorial there

2.4.2 Using Craftr for Cython projects

Craftr has convenient support for compiling Cython projects. The easy way is to use `compile_project()`.

```
from craftr import *
from craftr.ext.compiler import cython

cython.cythonc.compile_project(
  sources = path.glob('src/*.pyx'),
  python_bin = options.get('PYTHON', 'python'),
  additional_flags = ['-Xprofile=True'],
)
```

For more control, the Cython invocation and C/C++ source file compiling can be done manually. Below is the equivalent long version of the above shorthand:

```
# craftr_module(cython_test)
from craftr import *
from craftr.ext import platform, python
from craftr.ext.compiler import cython

# 1. Find the compilation information for the target Python version.
py = cython.PythonInfo(options.get('PYTHON', 'python'))

# 2. Compile the .pyx files to C-files.
pyxc_sources = cython.cythonc.compile(
  py_sources = path.glob('src/*.pyx'),
  python_version = py.major_version,
  cpp = False,
  additional_flags = ['-Xprofile=True']
)

# 3. Compile each C file to a shared library.
for pyxfile, cfile in zip(pyxc_sources.inputs, pyxc_sources.outputs):
  platform.ld.link(
    output = path.setsuffix(pyxfile, py.conf['SO']),
    output_type = 'dll',
    keep_suffix = True, # don't let link() replace the suffix
    inputs = platform.cc.compile(
      sources = [cfile],
      frameworks = [py.fw],
      pic = True
    )
  )
```

Compiling with `--embed`

Cython has an `--embed` command-line option that will cause the generated C/C++ source code to contain a `main()` entry point. You can just pass the `main` parameter to `compile_project()` and it will automatically generate an executable:

```
from craftr import *
from craftr.ext import rules
from craftr.ext.compiler import cython

project = cython.cythonc.compile_project(
  main = path.local('main.pyx'),
  python_bin = options.get('PYTHON', 'python'),
)

# Allows you to invoke `craftr .run` to compile and run
run = rules.run(project.main_bin)
```

Note: You can combine compiling C-Extensions and an executable in a single call to *`compile_project()`*.

2.4.3 Writing a Compiler Plugin

Craftr does not provide you with "one way to do it". There are multiple ways you can make Craftr generate the command you need it to. You can hard-code the command by creating a *`Target`* from scratch or you can implement a *Generator Function*. What we do most of the time is to implement a *Compiler Class* which inherits `craftr.ext.compiler._base.BaseCompiler`. It allows us to create instances of "compiler interfaces" with different settings, which makes these settings included in all procedures that generate targets.

Manual Targets

First things first though, here's a small example how you can just manually create a target and have Craftr export that into the Ninja manifest:

```
from craftr import path, Target

main = Target(
  command = 'gcc $in -Wall -std=c++11 -o $out',
  inputs = path.glob('src/*.c'),
  outputs = ['main'],
)
```

Notice how we specify just plain `'main'` as the output file: relative filenames will be considered relative to the build directory! Craftr automatically and *always* changes the working directory to the build directory before executing any code.

Generator Functions

Given the above simple GCC example, we can make things a bit more customizable by implementing a function that generates the command and target for us.

```
from craftr import path, Target

def compile(sources, output, include=[], defines=[],
```

```
            lib=[], libpath=[], warn='1', std='c99'):
  command = ['gcc', '$in', '-W' + warn, '-std=' + std)
  command += ['-I' + x for x in include]
  command += ['-D' + x for x in defines]
  command += ['-L' + x for x in libpath]
  command += ['-l' + x for x in lib]
  return Target(command, sources, [output])

main = compile(
  sources = path.glob('src/*.c'),
  output = 'main',
  warn = 'all',
  std = 'c++11'
)
```

Using the TargetBuilder

While the above example already looks nice, it still has problems, or say, complications: What will you do if you make use of some libraries and have a number of additional include directories, defines, libpaths and libs? Just concatenate them by hand?

Craftr's solution to this problem are *Framework* s. They represent a collection of settings that can either be merged (e.g. for things like include directories, defines, etc.) or the first available setting can be used (e.g. for some one-off compiler option). In Craftr, everything has frameworks. Just for example, a `Target` has a list of frameworks that have been used to generate it, thus if other targets are created taking it as an input, they can automatically re-use these frameworks and the user doesn't have to manually specify the framework yet another time.

```
from craftr.ext.platform import cc, ld
from craftr.ext.some_library import some_library_framework

obj = cc.compile(
  sources = path.glob('src/*.c'),
  frameworks = [some_library_framework]
)

bin = ld.link(
  inputs = obj,
  output = 'main'
  # <: Note how we do not add "some_library_framework" in this call
)
```

Moving on to creating *Target* generator functions with the *TargetBuilder*! This class handles a bunch of things, but don't let yourself be confused about all these internals yet. They are here for reference:

1. Evaluate a list of inputs that can consist of filenames or targets. Filenames are automatically normalized and for targets, the output files will be added to the input files and the frameworks will be included into the frameworks list.
2. Include a list of frameworks passed directly to the generator function.
3. Create a new `Framework` from the additional keyword arguments passed to the generator function, but this framework will **not** be included in the generated targets framework list! You don't want your `additional_flags` passed to `cc.compile()` also being passed to `ar.staticlib()` automatically :)
4. All frameworks will then be expanded into a single list using *expand_frameworks()* (to flatten out framework dependencies).

5. A *FrameworkJoin* will be created from *all* frameworks (including the special `**kwargs` framework) to enable the generator function to read the settings.

Now, how Tracer would say it, "let's get to it already!". Note that I've also added a `language` parameter which I did not in the previous examples.

```
from craftr import path, Target, TargetBuilder

def compile(sources, output, frameworks=(), target_name=None, language='c', **kwargs):
  builder = TargetBuilder(sources, frameworks, kwargs, name=target_name)
  include = builder.merge('include')
  defines = builder.merge('defines')
  libpath = builder.merge('libpath')
  lib = builder.merge('lib')
  std = builder.get('std', 'c99')
  warn = builder.get('warn', '1')

  # Same code as above
  command = ['gcc', '-x', language, '$in', '-W' + warn, '-std=' + std)
  command += ['-I' + x for x in include]
  command += ['-D' + x for x in defines]
  command += ['-L' + x for x in libpath]
  command += ['-l' + x for x in lib]

  return builder.create_target(command, output)

# Now we can use some other Craftfiles that expose Frameworks.
# (You know, Craftr's not really popular yet so there's literally
# only my own stuff right now :P)
from craftr.ext.libs.nr_iterator import nr_iterator
from craftr.ext.libs.nr_math3d import nr_math3d

main = compile(
  language = 'c++',
  sources = path.glob('src/*.cpp'),
  output = 'main',
  frameworks = [nr_iterator, nr_math3d]
)
```

Using the BaseCompiler

It has a number of advantages, but you're free to use a plain generator function as shown in the previous example! There's really not much to be changed for using a `BaseCompiler` instead:

```
from craftr import path, Target
from craftr.ext.compiler._base import BaseCompiler

class SimpleGCC(BaseCompiler):

  def compile(self, sources, output, frameworks=(), target_name=None, language='c', **kwargs):
    builder = self.builder(sources, frameworks, kwargs, name=target_name)
    # ... exactly the same code as in the previous example

gcc = SimpleGCC()
main = gcc.compile(
  # ...
)
```

However! you can now pass additional settings to the `SimpleGCC()` constructor that will be taken into account as well. Note that these are considered last after everything else (`**kwargs`, frameworks list, input target frameworks and only then the settings passed to the constructor).

Monkeypatching existing compilers

This is a technique that is used for instance by the maxon.c4d extension modules which requires additional preprocessing of the parameters passed to `cxx.compile()` and `ld.link()`. Since v1.1.1, the `BaseCompiler` supports hooking in after a *`TargetBuilder`* was created for a specific method call.

```
def _my_link_hook(builder):
  debug = builder.get('debug', options.get_bool('debug', False))
  builder.setdefault('output_type', 'dll')
  builder.add_framework(Framework('_my_link_hook',
    defines = ['_DEBUG'] if debug else ['NDEBUG'],
  ), local=True)

ld = platform.ld.fork()
ld.register_hook('link', _my_link_hook)
```

2.4.4 Automate build product distribution

Craftr provides a *`Archive`* class that can be used to easily create an archive of the products that are generated by the build, in the same step!

```
# craftr_module(test)
from craftr import *
from craftr.ext import platform, archive as _archive, git as _git
git = _git.Git(project_dir)

binary = platform.ld.link(
  inputs = platform.cc.compile(
    sources = path.glob('src/*.c')
  ),
  output = 'main'
)

@task(requires = [binary])
def archive():
  name = '{}-{}-{}'.format(project_name, git.describe(), platform.name)
  archive = _archive.Archive(prefix = name, base_dir = project_dir)
  archive.add(binary.outputs)
  archive.add('res')
  archive.save()
  info("Archive created:", path.normpath(archive.name, session.cwd))
```

Below you can find an example invokation of the script on Windows:

```
λ craftr .archive -v
detected ninja v1.6.0
cd "build"
load 'craftr.ext.test'
(craftr.ext.platform, line 74): Detected VS architecture: amd64
exporting 'build.ninja'
rts listening at 127.0.0.1:54411
$ ninja test.archive -v
```

```
[1/3] "C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin\amd64\cl.exe" /nologo /c c:\users\n
[2/3] "C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin\amd64\link.exe" /nologo c:\users\ni
[3/3] craftr-rts-invoke test.archive
(craftr.ext.:rts:): [127.0.0.1:54412] connection accepted
(craftr.ext.:rts:): [127.0.0.1:54412] @@ test.archive()
(craftr.ext.test, line 20): Archive created: c:\users\niklas\desktop\tes\test-v1.1-win.zip
```

2.5 API Documentation

This part of the documentation contains the API reference of the functions and classes that can be used in Craftfiles.

2.5.1 craftr.ext

class craftr.ext.**CraftrImporter**(*session*)

Meta-path import hook for importing Craftr modules from the *craftr.ext* parent namespace. Only functions inside a session context.

find_module(*fullname*, *path=None*)

PEP 0302 – New Import Hooks

import_file(*filename*)

Imports a Craftr module by *filename*. Raises *ImportError* if *filename* is not a Craftr module or if the file is not the same as would be imported when importing it by its module identifier.

update(*force=False*)

Should be called if *sys.path* or *Session.path* has been changed to rebuild the module cache and delay-load virtual modules if a physical was found.

class craftr.ext.**CraftrLoader**(*kind*, *filename*, *session*)

Loader class created by the :class:CraftrImporter.

load_module(*fullname*)

PEP 0302 – New Import Hooks

craftr.ext.**get_module_ident**(*filename*)

Extracts the module identifier from file at the specified *filename* and returns it, or None if the file does not contain a *craftr_module(...)* declaration in the first comment-block.

2.5.2 craftr.options

Utility functions to read options from the environment.

craftr.options.**get**(*name*, *default=None*, *inherit_global=True*)

Reads an option value from the environment variables. The option name will be prefixed by the identifier of the module that is currently executed, eg:

```
# craftr_module(test)
from craftr import options, environ
value = options.get('debug', inherit_global=False)
# is equal to
value = environ.get('test.debug')
```

Parameters

- **name** – The name of the option.

- **default** – The default value that is returned if the option is not set in the environment. If `NotImplemented` is passed for *default* and the option is not set, a `KeyError` is raised.
- **inherit_global** – If this is True, the option is also searched globally (ie. *name* without the prefix of the currently executed module).

Raises **KeyError** – If *default* is `NotImplemented` and the option does not exist.

craftr.options.**get_bool**(*name*, *default=False*, *inherit_global=True*)

Read a boolean option. The actual option value is interpreted as boolean value. Allowed values that are interpreted as correct boolean values are: `''`, `'true'`, `'false''`, ``''yes'`, `'no'`, `'0'` and `'1'`

Raises

- **KeyError** – If *default* is `NotImplemented` and the option does not exist.
- **ValueError** – If the option exists but has a value that can not be interpreted as boolean.

2.5.3 craftr.path

craftr.path.**addprefix**(*subject*, *prefix*)

Given a filename, this function will prepend the specified prefix to the base of the filename and return it. *filename* may be an iterable other than a string in which case the function is applied recursively and a list is being returned instead of a string.

__Important__: This is *not* the directy equivalent of *addsuffix()* as it considered *subject* to be a filename and appends the *prefix* only to the files base name.

craftr.path.**addsuffix**(*subject*, *suffix*, *replace=False*)

Given a string, this function appends *suffix* to the end of the string and returns the new string.

subject may be an iterable other than a string in which case the function will be applied recursively on all its items and a list is being returned instead of a string.

If the *replace* argument is True, the suffix will be replaced instead of being just appended. Make sure to include a period in the *suffix* parameter value.

craftr.path.**autoglob**(*path*, *parent=None*)

Returns *glob(path)* if *path* is actually a glob-style pattern. If it is not, it will return *[path]* as is, not checking wether it exists or not.

craftr.path.**buildlocal**(*path*)

Given a relative path, this function returns an absolute version assuming the path is relative to to current module's build directory.

Note: Can only be called from a module context (ie. from inside a Craftr module).

---

craftr.path.**commonpath**(*paths*)

Returns the longest sub-path of each pathname in the sequence *paths*. Raises *ValueError* if *paths* is empty or contains both relative and absolute pathnames. If there is only one item in *paths*, the parent directory is returned.

craftr.path.**get_long_path_name**(*path*)

On Windows, this function returns the correct capitalization for *path*. On all other systems, this returns *path* unchanged.

craftr.path.**glob**(**patterns*, *exclude=None*, *parent=None*)

Wrapper for *glob2.glob()* that accepts an arbitrary number of patterns and matches them. The paths are normalized with *normpath()*. If called from within a module, relative patterns are assumed relative to the modules parent directory.

If *exclude* is specified, it must be a string or a list of strings that is/contains glob patterns or filenames to be removed from the result before returning.

craftr.path.**isglob**(*path*)
Returns True if *path* is a glob-able pattern, False if not.

craftr.path.**iter_tree**(*dirname*, *depth=1*)
Iterates over all files in *dirname* and its sub-directories up to the specified *depth*. If *dirname* is a list, this scheme will be applied for all items in the list.

craftr.path.**listdir**(*path*, *abs=True*)
This version of *os.listdir* yields absolute paths.

craftr.path.**local**(*path*)
Given a path relative to the current module's project directory, this function returns a normalized absolute path. Just like many of the path functions, *path* can also be alist.

Note: Can only be called from a module context (ie. from inside a Craftr module).

---

craftr.path.**makedirs**(*path*)
Simple *os.makedirs()* clone that does not error if *path* is already an existing directory.

craftr.path.**move**(*filename*, *basedir*, *newbase*)
Given a filename and two directory names, this function generates a new filename which is constructed from the relative path of the filename and the first directory and the joint of this relative path with the second directory.

This is useful to generate a new filename in a different directory based on another. Craftr uses this function to generate object filenames.

Example:

```
>>> move('src/main.c', 'src', 'build/obj')
build/obj/main.c
```

path may be an iterable other than a string in which case the function is applied recursively to all its items and a list is returned instead of a string.

craftr.path.**normpath**(*path*, *parent_dir=None*, *abs=True*)
Normalizes a filesystem path. Also expands user variables. If a *parent_dir* is specified, a relative path is considered relative to that directory and converted to an absolute path. The default parent directory is the current working directory.

path may be an iterable other than a string in which case the function is applied recursively to all its items and a list is returned instead of a string.

If *abs* is True, the path is returned as an absolute path always, otherwise the path is returned in its original structure.

craftr.path.**relpath**(*path*, *start='.'*, *only_sub=False*)
Like the original *os.path.relpath()* function, but with the *only_sub* parameter. If *only_sub* is True and *path* is not a subpath of *start*, the *path* is returned unchanged.

craftr.path.**rmvsuffix**(*subject*)
Given a filename, this function removes the the suffix of the filename and returns it. If the filename had no suffix to begin with, it is returned unchanged.

subject may be an iterable other than a string in which case the function is applied recursively to its items and a list is returned instead of a string.

craftr.path.**setsuffix**(*subject*, *suffix*)
Remove the existing suffix from *subject* and add *suffix* instead. The *suffix* must contain the dot at the beginning.

craftr.path.**silent_remove**(*filename*, *is_dir=False*)
Remove the file *filename* if it exists and be silent if it does not. Returns True if the file was removed, False if it did not exist. Raises an error in all other cases.

Parameters

- **filename** – The path to the file or directory to remove.
- **is_dir** – If True, remove recursive (for directories).

craftr.path.**split_path**(*path*)
Splits *path* into a list of its parts.

class craftr.path.**tempfile**(*suffix=''*, *prefix='tmp'*, *dir=None*, *text=False*)
A better temporary file class where the `close()` function does not delete the file but only `__exit__()` does. Obviously, this allows you to close the file and re-use it with some other processing before it finally gets deleted.

This is especially important on Windows because apparently another process can't read the file while it's still opened in the process that created it.

```
from craftr import path, shell
with path.tempfile(suffix='c', text=True) as fp:
  fp.write('#include <stdio.h>\nint main() { }\n')
  fp.close()
  shell.run(['gcc', fp.name])
```

Parameters

- **suffix** – The suffix of the temporary file.
- **prefix** – The prefix of the temporary file.
- **dir** – Override the temporary directory.
- **text** – True to open the file in text mode. Otherwise, it will be opened in binary mode.

2.5.4 craftr.shell

This module is similar to the *subprocess.run()* interface that is available since Python 3.5 but is a bit customized so that it works better with Craftr.

class craftr.shell.**safe**
If this object is passed to *quote()*, it will not be escaped.

craftr.shell.**quote**(*s*)
Enhanced implementation for Windows systems as the original *shlex.quote()* function uses single-quotes on Windows which can lead to problems.

craftr.shell.**format**(*fmt*, **args*, ***kwargs*)
Similar to `str.format()`, but this function will escape all arguments with the *quote()* function.

craftr.shell.**join**(*cmd*)
Join a list of strings to a single command.

craftr.shell.**find_program**(*name*)
Finds the program *name* in the *PATH* and returns the full absolute path to it. On Windows, this also takes the *PATHEXT* variable into account.

Parameters **name** – The name of the program to find.

Returns `str` – The absolute path to the program.

Raises

- **FileNotFoundError** – If the program could not be found in the PATH.
- **PermissionError** – If a candidate for "name" was found but it is not executable.

craftr.shell.**test_program**(*name*)
Uses *find_program()* to find the path to "name" and returns True if it could be found, False otherwise.

exception craftr.shell.**CalledProcessError**(*process*)
This exception is raised when a process exits with a non-zero returncode and the run was to be checked for such state. The exception contains the process information.

exception craftr.shell.**TimeoutExpired**(*process*, *timeout*)
This exception is raised when a process did not exit after a specific timeout. If this exception was raised, the child process has already been killed.

class craftr.shell.**CompletedProcess**(*cmd*, *returncode*, *stdout*, *stderr*)
This class represents a completed process.

craftr.shell.**run**(*cmd*, ***, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *shell=False*, *timeout=None*, *check=False*, *cwd=None*, *encoding='utf-8'*)
Run the process with the specified *cmd*. If *cmd* is a list of commands and *shell* is True, the list will be automatically converted to a properly escaped string for the shell to execute.

Note: If "shell" is True, this function will manually check if the file exists and is executable first and raise FileNotFoundError if not.

Raises

- **CalledProcessError** – If *check* is True and the process exited with a non-zero exit-code.
- **TimeoutExpired** – If *timeout* was specified and the process did not finish before the timeout expires.
- **OSError** – For some OS-level error, eg. if the program could not be found.

craftr.shell.**pipe**(**args*, *merge=True*, ***kwargs*)
Like *run()*, but pipes stdout and stderr to a buffer instead of directing them to the current standard out and error files. If *merge* is True, stderr will be merged into stdout.

2.5.5 craftr.utils

Various common utilities used by Craftr and its extension modules.

Transform Functions

craftr.utils.**flatten**(*iterable*)
Given an *iterable* that in turn yields an iterable, this function flattens the nested iterables into a single iteration.

craftr.utils.**uniquify**(*iterable*)
Create a list of items in *iterable* without duplicate, preserving the order of the elements where it first appeared.

Recordclass

class craftr.utils.**recordclass_base**(**args*, ***kwargs*)
Base class that provides a namedtuple like interface based on the __slots__ parameter.

```
class MyRecord(recordclass_base):
  __slots__ = 'foo bar ham'.split()

data = MyRecord('a foo', 42, ham="spam")
```

items()

keys()

values()

craftr.utils.**recordclass**(*__name*, *__fields*, ***defaults*)
Creates a new class that can represent a record with the specified *fields*. This is equal to a mutable namedtuple. The returned class also supports keyword arguments in its constructor.

Parameters

- **__name** – The name of the recordclass.
- **__fields** – A string or list of field names.
- **defaults** – Default values for fields. The defaults may list field names that haven't been listed in *fields*.

Environment Variables

craftr.utils.**append_path**(*pth*)
Appends *pth* to the PATH environment variable.

craftr.utils.**prepend_path**(*pth*)
Prepends *pth* to the PATH environment variable.

craftr.utils.**override_environ**(*new_environ=None*)
Context-manager that restores the old environ on exit.

Parameters **new_environ** – A dictionary that will update the environ inside the context-manager.

craftr.utils.regex Module

Regex utility functions.

craftr.utils.regex.**search_get_groups**(*pattern*, *subject*, *mode=0*)
Performs re.search() and returns a list of the captured groups, *including* the complete matched string as the first group. If the regex search was unsuccessful, a list with that many items containing None is returned.

craftr.**session**
A Proxy to the current *Session* object that is being used for the current Craftr build session.

Note: If you've used Flask before: It's similar to the Flask request object.

---

craftr.**module = <Proxy unbound>**
This *werkzeug.LocalProxy* subclass returns the current object when called instead of forwarding the call to the current object.

A `Proxy` of the Craftr module that is currently being executed. Modules are standard Python module objects. When a Craftr extension module is being executed, this proxy points to exactly that module.

```
# craftr_module(test)
# A stupid example
from craftr import module
import sys
assert project_name == module.project_name
assert sys.modules[__name__] is module()
```

2.5.6 Logging

The logging functions implement the `print()` interface.

craftr.**debug**(**args*, *stacklevel=1*, *verbosity=None*, ***kwargs*)

craftr.**info**(**args*, *stacklevel=1*, ***kwargs*)

craftr.**warn**(**args*, *stacklevel=1*, ***kwargs*)

craftr.**error**(**args*, *stacklevel=1*, *raise_=True*, ***kwargs*)

2.5.7 Tasks

craftr.**task**(*func=None*, **args*, ***kwargs*)
Create a task *Target* that uses the Craftr RTS feature. If *func* is None, this function returns a decorator that finally creates the *Target*, otherwise the task is created instantly.

The wrapped function must either

- take no parameters, this is when both the *inputs* and *outputs* of the task are `None`, or
- take two parameters being the *inputs* and *outputs* of the task

```
@task
def hello():  # note: no parameters
  info("Hello, World!")

@task(inputs = another_target, outputs = 'some/output/file')
def make_some_output_file(inputs, outputs):  # note: two parameters!
  # ...

yat = task(some_function, inputs = yet_another_target,
           name = 'yet_another_task')
```

Important: Be aware that tasks executed through Ninja (and thus via RTS) are executed in a seperate thread!

Note that unlike normal targets, a task is explicit by default, meaning that it must explicitly be specified on the command line or be required as an input to another target to be executed.

Parameters

- **func** – The callable function to create the RTS target with or None if you want to use this function as a decorator.

- **args** – Additional args for the *Target* constructor.
- **kwargs** – Additional kwargs for the *Target* constructor.

Returns *Target* or a decorator that returns *Target*

2.5.8 Helpers

craftr.**return_**()
Raise a ModuleReturn exception, causing the module execution to be aborted and returning back to the parent module. Note that this function can only be called from a Craftr modules global stack frame, otherwise a RuntimeError will be raised.

craftr.**expand_inputs**(*inputs*, *frameworks=None*)
Expands a list of inputs into a list of filenames. An input is a string (filename) or a *Target* object from which the *Target.outputs* are used. Returns a list of strings.

If *frameworks* is specified, it must be a list to which the frameworks of all input *Target* objects will be appended. The frameworks need to be expanded with *expand_frameworks()*.

craftr.**expand_frameworks**(*frameworks*, *result=None*)
Given a list of *Framework* objects, this function creates a new list that contains all objects of *frameworks* and additionally all objects that are listed in each of the frameworks "frameworks" key recursively. Duplicates are also elimated.

craftr.**import_file**(*filename*)
Import a Craftr module by filename. The Craftr module identifier must be determinable from this file either by its #craftr_module(..) identifier or filename.

craftr.**import_module**(*modname*, *globals=None*, *fromlist=None*)
Similar to importlib.import_module(), but this function can also imports contents of *modname* into *globals*. If *globals* is specified, the module will be directly imported into the dictionary. If *fromlist* list is *, a wildcard import into *globals* will be performed, otherwise *fromlist* must be None or a list of names to import.

This function always returns the root module.

craftr.**craftr_min_version**(*version_string*)
Ensure the current version of Craftr is at least the version specified with *version_string*, otherwise call *error()*.

2.5.9 Session Objects

class craftr.**Session**(*cwd=None*, *path=None*, *server_bind=None*, *verbosity=0*, *strace_depth=3*, *export=False*, *buildtype='standard'*)
This class manages a build session and encapsulates all Craftr modules and *Targets*.

cwd
The original working directory from which Craftr was invoked, or the directory specified with the -p command-line option. This is different than the current working directory since Craftr changes to the build directory immediately.

env
A dictionary of environment variables, initialized as a copy of os.environ. In a Craftfile, you can use os.environ or the alias craftr.environ instead, which is more convenient than accessing session.env.

path
A list of search paths for Craftr extension modules. See ext.

modules
A dictionary of Craftr extension modules. Key is the module name without the `craftr.ext.` prefix.

targets
A dictionary mapping the full identifier to *Target* objects that have been declared during the build session. When the Session is created, a `clean` Target which calls `ninja -t clean` is always created automatically.

files_to_targets
New in v1.1.0 Maps the files produced by all targets to their producing *Target* object. This dictionary is used for speeding up *find_target_for_file()* and to check if any file would be produced by multiple targets.

All keys in this dictionary are absolute filenames normalized with *path.normpath()*.

server
An `rts.CraftrRuntimeServer` object that is started when the session context is entered with `magic.enter_context()` and stopped when the context is exited. See *on_context_enter()*.

server_bind
A tuple of `(host, port)` which the *server* will be bound to when it is started. Defaults to None, in which case the server is bound to the localhost on a random port.

ext_importer
A *ext.CraftrImporter* object that handles the importing of Craftr extension modules. See `ext`.

var
A dictionary of variables that will be exported to the Ninja manifest.

verbosity
The logging verbosity level. Defaults to 0. Used by the logging functions *debug()*, *info()*, *warn()* and *error()*.

strace_depth
The logging functions may print a stack trace of the log call when the verbosity is high enough. This defines the depth of the stack trace. Defaults to 3.

export
This is set to True when the `-e` option was specified on the command-line, meaning that a Ninja manifest will be exported. Some projects eventually need to export additional files before running Ninja, for example with *TargetBuilder.write_command_file()*.

buildtype
The buildtype that was specified with the `--buildtype` command-line option. This attribute has two possible values: `'standard'` and `'external'`. Craftfiles and rule functions must take the buildtype into consideration. In `'external'` mode, rule functions should consider external options wherever applicable, for example the `CFLAGS` environment variables instead or additionally to the standard flags for C source file compilation.

finalized
True if the Session was finalized with *finalize()*.

exec_if_exists(*filename*)
Executes *filename* if it exists. Used for running the Craftr environment files before the modules are loaded. Returns None if the file does not exist, a *types.ModuleType* object if it was executed.

finalize()
Finalize the session, setting up target dependencies based on their input/output files to simplify verifying dependencies inside of Craftr. The session will no longer accept target registrations.

find_target_for_file(*filename*)
Finds a target that outputs the specified *filename*.

on_context_enter(*prev*)
Called when entering the Session context with `magic.enter_context()`. Does the following things:

- Sets up the `os.environ` with the values from *Session.env*
- Adds the *Session.ext_importer* to `sys.meta_path`

Note: A copy of the original `os.environ` is saved and later restored in *on_context_leave()*. The `os.environ` object *can not* be replaced by another object, that is why we change its values in-place.

---

on_context_leave()
Called when the context manager entered with `magic.enter_context()` is exited. Undos all of the stuff that *on_context_enter()* did and more.

- Stop the Craftr Runtime Server if it was started
- Restore the `os.environ` dictionary
- Removes all `craftr.ext.` modules from `sys.modules` and ensures they are in *Session.modules* (they are expected to be put there from the *ext.CraftrImporter*).

register_target(*target*)
This function is used by the *Target* constructor to register itself to the *Session*. This will add the target to the `target` dictionary and also update the *files_to_targets* mapping.

Parameters **target** – A *Target* object

Raises

- **ValueError** – If the name of the target is already reserved.
- **RuntimeError** – If this target produces a file that is already produced by another arget.

start_server()
Start the Craftr RTS server (see *Session.server*). It will automatically be stopped when the session context is exited.

2.5.10 Target Objects

class craftr.**Target**(*command*, *inputs=None*, *outputs=None*, *implicit_deps=None*, *order_only_deps=None*, *requires=None*, *foreach=False*, *description=None*, *pool=None*, *var=None*, *deps=None*, *depfile=None*, *msvc_deps_prefix=None*, *explicit=False*, *frameworks=None*, *meta=None*, *module=None*, *name=None*)
This class is a direct representation of a Ninja rule and the corresponding in- and output files. Will be rendered into a `rule` and one or many `build` statements in the Ninja manifest.

New in v1.1.0: A target object can also represent a Python function as a target in the Ninja manifest. This is called an RTS task. Use the *task()* function to create tasks or pass a function for the *command* parameter of the *Target* constructor. The function must accept no parameters if *inputs* and *outputs* are **both** `None` or accept these two values as parameters.

name
The name of the target. This is usually deduced from the variable the target is assigned to if no explicit name was passed to the *Target* constructor. Note that the actual name of the generated Ninja rule must be read from *fullname*.

module
The Craftr extension module this target belongs to. Defaults to the currently executed module (retrieved from the thread-local *module*). Can be None, but only if there is no module currently being executed.

command
A list of strings that represents the command to execute. A string can be passed to the constructor in which case it is parsed with `shell.split()`.

inputs
A list of filenames that are listed as inputs to the target and that are substituted for `$in` and `$in_newline` during the Ninja execution. Can be None. The *Target* constructor expands the passed argument with *expand_inputs()*, thus also accepts a single filename, Target or a list with Targets and/or filenames.

This attribute can also be None.

outputs
A list of filenames that are listed as outputs of the target and that are substituted for `$out` during the Ninja execution. Can be None. The *Target* constructor accepts a list of filenames or a single filename for this attribute.

This attribute can also be None.

implicit_deps
A list of filenames that are required to build the Target, additionally to the *inputs*, but are not expanded by the `$in` variable in Ninja. See "Implicit dependencies" in the Ninja Manual.

order_only_deps
See "Order-only dependencies" in the Ninja Manual.

requires
A list of targets that are to be built before this target is. This is useful for speciying task dependencies that don't have input and/or output files.

The constructor accepts None, a *Target* object or a list of targets and will convert it to a list of targets.

```
@task
def hello():
  info("Hello!")

@task(requires = [hello])
def ask_name():
  info("What's your name?")
```

foreach
If this is set to True, the number of *inputs* must match the number of *outputs*. Instead of generating a single `build` instruction in the Ninja manifest, an instruction for each input/output pair will be created instead. Defaults to False.

description
A description of the Target. Will be added to the generated Ninja rule. Defaults to None.

pool
The name of the build pool. Defaults to None. Can be `"console"` for Targets that don't actually build files but run a program. Craftr will treat Targets in that pool as if *explicit* is True.

deps
The mode for automatic dependency detection for C/C++ targets. See the "C/C++ Header Depenencies" section in the Ninja Manual.

depfile
A filename that contains additional dependencies.

msvc_deps_prefix
The MSVC dependencies prefix to be used for the rule.

frameworks
A list of *Frameworks* that are used by the Target. Rule functions that take other Targets as inputs can include this list. For example, a C++ compiler might add a Framework with `libs = ['c++']` to a Target so that the Linker to which the C++ object files target is passed automatically knows to link with the `c++` library.

Usually, a rule function uses the *TargetBuilder* (which internally uses *expand_inputs()*) to collect all Frameworks used in the input targets.

explicit
If True, the target will only be built by Ninja if it is explicitly specified on the command-line or if it is required by another target. Defaults to False.

meta
A dictionary of meta variables that can be set from anywhere. Usually, rule functions use this dictionary to promote additional information to the caller, for example what the actual computed output filename of a compilation is.

graph
Initially None. After *finalize()* is called, this is a namedtuple of *Graph* which has input and output sets of targets of the dependencies in the Target.

__lshift__(*other*)
Shift operator to add to the list of *implicit_deps*.

Note: If *other* is or contains a *Target*, the targets frameworks are *not* added to this Target's framework list!

---

class Graph(*inputs*, *outputs*)
Type for *Target.graph*

inputs
Alias for field number 0

outputs
Alias for field number 1

`Target`.**RTS_Mixed = 'mixd'**
The target and/or its dependencies are a mix of command-line targets and tasks

`Target`.**RTS_None = 'none'**
The target and its dependencies are plain command-line targets

`Target`.**RTS_Plain = 'plain'**
The target and all its dependencies are plain task targets

`Target`.**as_explicit**()
Sets :attr'explicit' to `True` and retunrs *self*.

`Target`.**execute_task**(*exec_state=None*)
Execute the `rts_func` of the target. This calls the function with the inputs and outputs of the target (if any of these are not None) or with no arguments (if both is None).

This function catches all exceptions that the wrapped function might raise and prints the traceback to stdout and raises a `TaskError` with status-code 1.

Parameters **exec_state** – If this parameter is not None, it must be a dictionary where the task can check if it already executed. Also, inputs of this target will be executed if the parameter is a dictionary.

Raises

- **RuntimeError** – If the target is not an RTS task.
- **TaskError** – If this task (or any of the dependent tasks, only if *exec_state* is not None) exits with a not-None, non-zero exit code.

Target.**finalize**(*session*)
Gather the inputs and outputs of the target and create a new *Graph* to fill the *graph* attribute.

Target.**fullname**
The full identifier of the Target. If the Target is assigned to a *module*, this is the module name and the *Target.name*, otherwise the same as *Target.name*.

Target.**get_rts_mode**()
Returns the RTS information for this target:

- *RTS_None* if this target and none of its dependencies
- *RTS_Plain* if this target and all of its dependencies are tasks
- *RTS_Mixed* if this target or any of its dependencies are tasks but there is at least one normal target

2.5.11 TargetBuilder Objects

class craftr.**TargetBuilder**(*inputs*, *frameworks=()*, *kwargs=None*, *meta=None*, *module=None*, *name=None*, *stacklevel=1*)
This is a helper class to make it easy to implement rule functions that create a *Target*. Rule functions usually depend on inputs (being files or other Targets that can also contain additional frameworks), rule-level settings and *Frameworks*. The TargetBuilder takes all of this into account and prepares the data conveniently.

The following example shows how to make a simple rule function that compiles C/C++ source files into object files with GCC. The actual compiler name can be overwritten and additional flags can be specified by passing them directly to the rule function or via frameworks (accumulative).

```
#craftr_module(test)

from craftr import TargetBuilder, Framework, path
from craftr.ext import platform
from craftr.ext.compiler import gen_output

def compile(sources, frameworks=(), **kwargs):
  """
  Simple rule to compile a number of source files into an
  object files using GCC.
  """

  builder = TargetBuilder(sources, frameworks, kwargs)
  outputs = gen_output(builder.inputs, suffix = platform.obj)
  command = [builder.get('program', 'gcc'), '-c', '$in', '-o', '$out']
  command += builder.merge('additional_flags')
  return builder.create_target(command, outputs = outputs)

copts = Framework(
  additional_flags = ['-pedantic', '-Wall'],
)
```

```
objects = compile(
  sources = path.glob('src/**/*.c'),
  frameworks = [copts],
  additional_flags = ['-std=c11'],
)
```

Parameters

- **inputs** – Inputs for the target. Processed by *expand_inputs()*, the resulting frameworks are then processed by *expand_frameworks()*. The expanded inputs are saved in the *inputs* attribute of the *TargetBuilder*. Use this attribute instead of the original value passed to this parameter! It is guaruanteed to be a list of filenames only.
- **frameworks** – A list of frameworks to take into account additionally.
- **kwargs** – Additional options that will be turned into their own *Framework* object, but it will *not* be passed to the Target that is created with *create_target()* as these options should not be inherited by rules that will receive the target as input.
- **module** – Override the module that will receive the target.
- **name** – Override the target name. If not specified, the target name is retrieved using Craftr's target name deduction from the name the target is assigned to.
- **stacklevel** – The stacklevel which the calling rule function is at. This defaults to 1, which is fine for rule functions that directly create the *TargetBuilder*.

caller
Name of the calling function.

```
def my_rule(*args, **kwargs):
  builder = TargetBuilder(None)
  assert builder.caller == 'my_rule'
```

inputs
`None` or a pure list of filenames that have been passed via the *inputs* parameter of the TargetBuilder.

frameworks
A list of frameworks compiled from the frameworks of *Target* objects in the *inputs* parameter of the constructor and the frameworks that have been specified directly with the *frameworks* parameter.

kwargs
The additional options that have been passed with the *kwargs* argument. These are turned into their own *Framework* which is only taken into account for the *options* but it is not passed to the *Target* created with *create_target()*.

options
A *FrameworkJoin* object that is used to read settings from the list of frameworks collected from the input Targets, the additional frameworks specified to the *TargetBuilder* constructor and the specified `kwargs` dictionary.

module

name
The name of the Target that is being built.

target_attrs
A dictonary of arguments that are set to the target after construction in *create_target()*. Can only set attributes that are already attributes of the *Target*.

meta
Meta data for the Target that is passed directly to *Target.meta*.

__getitem__(*key*)
Alias for `FrameworkJoin.__getitem__()` on the *options*.

add_framework(*fw*, *local=False*, *front=False*)
Adds the *Framework* "fw" to the builder and also to the target if "local" is False. The framework will be appended to the end of the chain, thus is has the lowest priority unless you pass "front" to True.

Parameters

- **fw** – The framwork to add.
- **local** – If this is False, the framework will also be added to the target created by the builder.
- **front** – If this is True, the framework will be added to the front of the frameworks list and thus will be treated with high priority.

create_target(*command*, *inputs=None*, *outputs=None*, ***kwargs*)
Create a *Target* and return it.

Parameters

- **command** – The command-line for the Target.
- **inputs** – The inputs for the Target. If None, the *TargetBuilder.inputs* will be used instead.
- **outputs** – THe outputs for the Target.
- **kwargs** – Additional keyword arguments for the Target constructor. Make sure that none conflicts with the *target* dictionary.

Note: This function will yield a warning when there are any keys in the *kwargs* dictionary that have not been read from the *options*.

---

expand_inputs(*inputs*)
Wrapper for *expand_inputs()* that will add the Frameworks extracted from the *inputs* to *options* and *frameworks*.

fullname
The full name of the Target that is being built.

get(*key*, *default=None*)
Alias for *FrameworkJoin.get()*.

invalid_option(*option_name*, *option_value=<object object>*, *cause=None*)
Use this method in a rule function if you found the value of an option has an invalid option. You should raise a `ValueError` on a fatal error instead.

log(*level*, **args*, *stacklevel=1*, ***kwargs*)
Log function that includes the *fullname*.

merge(*key*)
Alias for *FrameworkJoin.merge()*.

mkname(*name*)
Create a unique target identifier which based on this target builders *name* and an incrementing index.

setdefault(*key*, *value*)
Sets a value in the `fwdefaults` framework.

target
A dictonary of arguments that are set to the target after construction in *create_target()*. Can only set attributes that are already attributes of the *Target*.

Deprecated since version Use: *target_attrs* instead.

write_command_file(*arguments*, *suffix=None*, *always=False*)
Writes a file to the `CMDDIR` folder in the build directory (ie. the current directory) that contains the command-line arguments specified in *arguments*. The name of that file is the name of the Target that is created with this builder. Optionally, a suffix for that file can be specified to be able to write multiple such files. Returns the filename of the generated file. If *always* is set to True, the file will always be created even if *Session.export* is set to False.

write_multicommand_file(*commands*, *cwd=None*, *exit_on_error=True*, *suffix=None*, *always=False*)
Write a platform dependent script that executes the specified *commands* in order. If *exit_on_error* is True, the script will exit if an error is encountered while executing the commands.

Returns a list representing the command-line to run the script.

Parameters

- **commands** – A list of strings or command lists that are written into the script file.
- **cwd** – Optionally, the working directory to change to when the script is executed.
- **exit_on_error** – If this is True, the script will exit immediately if any command returned a non-zero exit code.
- **suffix** – An optional file suffix. Note that on Windows, `.cmd` is added to the filename after that suffix.
- **always** – If this is true, the file is always created, not only if a Ninja manifest is being exported (see *Session.export*).

Returns A tuple of two elements. The first element is a command list that represents the command used to invoke the created script. The second element is the actual command file that was written.

2.5.12 Framework Objects

class `craftr.`**Framework**(*_Framework__fw_name=None*, *_Framework__init_dict=None*, ***kwargs*)
A Framework represents a set of options that are to be taken into account by compiler classes. Eg. you might create a framework that contains the additional information and options required to compile code using OpenCL and pass that to the compiler interface.

Compiler interfaces may also add items to *Target.frameworks* that can be taken into account by other target rules. *expand_inputs()* returns a list of frameworks that are being used in the inputs.

Use the *FrameworkJoin* class to create an object to process the data from multiple frameworks.

Parameters

- **__fw_name** – The name of the Framework. If omitted, the assigned name of the calling module will be used.
- **__init_dict** – A dictionary to initialize the Framework with.
- **kwargs** – Additional key/value pairs for the Framework.

2.5.13 FrameworkJoin Objects

class craftr.**FrameworkJoin**(**frameworks*)

This class is used to process a set of *Frameworks* and retreive relevant information from it. For some options, you might want to read the first value that is specified in any of the frameworks, for another you may want to create a list of all values in the frameworks. This is what the FrameworkJoin allows you to do.

Note: The *FrameworkJoin* does not use *expand_frameworks()* but uses the list of frameworks passed to the constructor as-is.

```
>>> fw1 = Framework('fw2', defines=['DEBUG'])
>>> fw2 = Framework(defines=['DO_STUFF'])
>>> print(fw2.name)
'fw2'
>>> FrameworkJoin(fw1, fw2).merge('defines')
['DEBUG', 'DO_STUFF']
```

used_keys

A set of keys that have been accessed via __getitem__(), *get()* and *merge()*.

frameworks

The list of *Framework* objects.

defaults

An additional framework that can be used to set default values. This framework will always be checked last.

__iadd__(*frameworks*)

get(*key*, *default=None*)

Get the first available value of *key* from the frameworks.

keys()

Returns a set of all keys in all frameworks.

merge(*key*)

Merge all values of *key* in the frameworks into one list, assuming that every key is a non-string sequence and can be appended to a list.

2.6 Craftr's Python Magic

Craftr uses some magic tricks behind the scenes to make the interface as convenient as possible. Most of the magic comes from the craftr.magic module!

2.6.1 Proxies

Craftr uses the werkzeug.local module to provide the *craftr.session* and *craftr.module* proxies that represent the current session and currently executed module respectively. This is how the *craftr.Target* constructor (and subsequently all functions that create a Target) knows in what module the target is being declared.

2.6.2 Target Name Deduction

Target names are automatically deduced from the variable name that the declared target is assigned to. This is enabled by parsing the bytecode of the global stackframe of the current module. This is more convenient that writing the name of the target twice by passing the `name` parameter to the *`craftr.Target`* constructor or a rule function.

```
objects = Target(
  command = 'gcc $in -o $out -c',
  inputs = sources,
  outputs = objects,
)
assert objects.name == 'objects'
```

Check the `craftr.magic.get_assigned_name()` function for details on the implementation of this feature.

2.6.3 Craftr RTS

The Craftr Runtime Server is a socket server that is started on a random port on the localhost when Craftr is started. The `craftr-rts-invoke` command can connect to that server and execute Python functions in the original Craftr process. The address of the server is saved in the `CRAFTR_RTS` environment variable. There are a few limitations to this method:

- The execution phase can not be skipped when RTS is required
- You can not pipe into the `craftr-rts-invoke` script

2.7 Changelog

2.7.1 v1.1.1

- Bug fixes
  - Logging in Craftr RTS fails with Runtime Error (#104)
- Behaviour changes
  - add `__no_default` target when there are no default targets, printing "no default target"
  - removed default `clean` target, use `-c` or `-cc` command-line option
  - catching `craftr.ModuleError` no longer prints the error text (#118)
- API related changes
  - add `frame` and `module` argument to `craftr.log()`
  - add `Target.as_explicit()`
  - add *`craftr.ext.platform.asm`* compiler proxy
  - `craftr.memoize_tool()` will be deprecated in the future and is now a synonym for `functools.lru_cache()`
  - *`craftr.shell.run()`* now manually checks if the program exists and raises a `FileNotFoundError` exception if it does not (only if `shell=True`)
  - add *`craftr.utils.override_environ()`*
  - add *`craftr.ext.rules.alias()`* function

  - add *craftr.TargetBuilder.mkname()* method
  - add *craftr.TargetBuilder.setdefault()* method
  - add *craftr.FrameworkJoin.defaults* member
  - add craftr.FrameworkJoin.iter_frameworks() method
  - moved craftr.ext.compiler.BaseCompiler to *craftr.ext.compiler.base.BaseCompiler*, backwards compatible import exists
  - removed BaseCompiler.__getitem__() and ~.__setitem__()
  - add BaseCompiler.register_hook()
  - *craftr.TargetBuilder.add_framework()* was updated
  - replace craftr.utils.slotobject() with *recordclass()* (alias introduced for backwards compatibility)
  - *craftr.utils* is now a package, some name changes but backwards compatibility has been kept by introducing aliases
  - fix Proxy __name__ attribute always returning None instead of the underlying object's member value
  - fix *craftr.path.buildlocal()* now using project_name instead of __name__
  - *cc*, *cxx*, *ld* etc. are no longer proxies but real objects
- C/C++ related changes
  - C/C++ compiler implementations now take debug option into account if no explicit value is passed to the generator function
  - removed 'clang' as a compiler name
  - added support for ***_compile_remove_flags and ***_link_remove_flags options where *** can be msvc, llvm and gcc
  - add support for msvc_runtime_library and force_include options
  - add support for link_target output variable
- Cython related changes
  - add Cython tutorial to docs
  - Cython compiler program can now be overwritten with CYTHONC
  - add support for embed parameter to *compile()*
  - add *PythonInfo* class
  - add *compile_project()* method
- *craftr.ext.cmake*
  - renamed render_config() to *configure_file()* to match the CMake naming and update parameter names

2.7.2 v1.1.0

- NEW: Tasks (replaces craftr.ext.rules.PythonTool)
  - created with the new task() function/decorator
  - can be specified on the command-line

  - exported to the Ninja manifest
  - run through Craftr RTS
- huge file naming scheme changes (issue #95)
  - rename `Craftfile` to `Craftfile.py`
  - rename `.craftrc` to `craftrc.py`
  - rename `<some_module>.craftr` to `craftr.ext.<some_module>.py`
- Standard Library
  - remove `craftr.ext.options` module, use `craftr.options` instead (issue #97)
  - add support for `msvc_runtime_library_option` which can have the value `'dynamic'` or `'static'`
  - remove `craftr.ext.rules.PythonTool` and rewrite `~.render_template()`
  - update `compiler.cython` documentation
  - fix missing `foreach=True` in `CythonCompiler.compile()`
  - add *craftr.ext.python* module
  - fix `-shared` argument to LLVM/GCC `.link()` rule (fix #109)
  - MSVC C++ compiler is now read from `CXX` variable instead of `CC`
  - Linux linker is now read from `CC` variable instead of `CCLD`
  - support for `CFLAGS`, `CPPFLAGS`, `ASMFLAGS`, `LDFLAGS` and `LDLIBS` (see issue #111)
  - Add `craftr.ext.cmake` module (issue #113)
- General
  - `setup.py` now uses `entry_points` to install console scripts (issue #94)
- Behaviour changes
  - automatically import targets specified on the command-line (issue #96)
  - catch possible PermissionError in `CraftrImporter._rebuild_cache()` (sha 16a6e307)
  - module and session context is now available when a task is executed (issue #99)
  - fix `TargetBuilder.write_command_file()`, now correctly returns the filename even if no file is actually created
  - sophisticated target check on build-only invokation if RTS is required (and thus the execution step can not be skipped) (issue #98)
  - new Craftr data caching method using JSON in the Ninja build manifest (also fixes #100) (issue #101)
  - Craftr RTS now works with task-targets, removed `MSG_ARGUMENT` and `_RequestHandler.arglist`
  - functions wrapped with the `task()` decorator can now be specified on the command-line just like normal targets (due to the fact that they are real targets also exported to the Ninja manifest)
  - if all targets specified on the command-line are tasks and do not depend on Ninja-buildable targets, the task(s) will be executed without Ninja (issue #103)
  - if `-e` is not specified but the manifest does not exist, export will be forced unless the specified targets do not require it (ie. are plain tasks) (see #103)

  - calling `Session.update()` after altering `Session.path` is no longer necessary (issue #108)

- Command-line changes
  - inverted behaviour of `-e`!! Now causes skip of the export and eventually execution step (if possible), short version of `--skip-export`
  - inverted behaviour of `-b`!! Now causes skip of the build phase, short version for `--skip-build`
  - removed `-f` and `-F` command-line options completely (instead, tasks that do not depend on normal targets can be executed without Ninja, see #103)
  - deprecated `-b` flag, the build step is now always executed by default
  - add `-n` flag which is the inverse of the old `-b` flag, skip the build phase if specified
  - updated command help
  - passing `-v` will automatically add `-v` to the Ninja invokation
  - add `--buildtype` option for which you can choose to pass the value `standard` (default) or `external`
- API Changes
  - add `task()` decorator function
  - add `TaskError` exception class
  - `TargetBuilder()` now accepts None for its *inputs* parameter
  - `TargetBuilder()` now has default values for the *frameworks* and *kwargs* parameters
  - removed `options.get_option()`
  - `options.get()` now accepts a *default* parameter, updated its docstrings
  - passing `NotImplemented` for *default* to `options.get()` now raises a `KeyError` if the option does not exist
  - add `option.get_bool()`
  - removed `Session.update()` (see issue #108)
  - removed `Session.rts_funcs`
  - add `Session.files_to_targets`
  - add `Session.finalized`
  - add `Session.finalize()`
  - add `Session.find_target_for_file()`
  - add `Session.buildtype`
  - add `Target.rts_func`
  - add `Target.requires`
  - add `Target.graph`
  - add `Target.finalize`
  - add `Target.finalized` property
  - add `Target.get_rts_mode()`
  - add `Target.execute_task()`

  - Targets can now also be tasks which will be executed through Craftr RTS by passing a callable to the constructor for the *command* argument (you should prefer the `task()` function though)
  - add `craftr.path.buildlocal()` function
  - add `craftr.shell.format()` and `~.join()` functions
  - `craftr.shell.run()` now splits strings into a command list if the *shell* argument is False
- Logging
  - removed the `craftr:  [INFO ]:` prefix stuff
  - logging functions only display the source module when at least `-v` is specified
  - updated output coloring and debug message strings
  - stracktrace for log entries now skips builtin modules

2.7.3 v1.0.0

- initial release version

CHAPTER 3

Getting Started

Craftr is built around Python-ish modules that we call Craftr modules or Craftfiles (though this name usually refers to the first type of Craftr modules). There are two ways a Craftr module can be created:

1. A file named `Craftfile.py` with a `# craftr_module(...)` declaration
2. A file named `craftr.ext.<module_name>.py` where `<module_name>` is of course the name of your Craftr module

By default, Craftr will execute the `Craftfile.py` from the current working directy if no different main module is specified with the `-m` option. Below you can find a simple Craftfile that can build a C++ program on any platform (that is supported by the Craftr STL).

```
# craftr_module(my_project)
from craftr import path
from craftr.ext import platform

# Create object files for each .cpp file in the src/ directory.
obj = platform.cxx.compile(
  sources = path.glob('src/*.cpp'),
  std = 'c++11',
)

# Link all object files into an executable called "main".
program = platform.ld.link(
  inputs = obj,
  output = 'main'
)
```

Below is a sample invokation on Windows. We pass the `-v` flag for additional debug output by Craftr and full command-line output from Ninja.

```
λ craftr -v
detected ninja v1.6.0
cd "build"
load 'craftr.ext.my_project'
(craftr.ext.my_project, line 9): unused options for compile(): {'std'}
exporting 'build.ninja'
$ ninja -v
[1/2] cl /nologo /c c:\users\niklas\desktop\test\src\main.cpp /Foc:\users\niklas\desktop\test\build\m
[2/2] link /nologo c:\users\niklas\desktop\test\build\my_project\obj\main.obj /OUT:c:\users\niklas\de

λ ls build build\my_project\
build:
build.ninja  my_project/
```

```
build\my_project\:
main.exe*  obj/
```

CHAPTER 4

Installation

```
pip install craftr-build
```

To install from the Git repository, use the `-e` flag to be able to update Craftr by simply pulling the latest changes from the remote repository.

```
git clone https://github.com/craftr-build/craftr.git && cd craftr
pip install -e .
```

CHAPTER 5

Targets

Craftr describes builds with the *craftr.Target* class. Similar to rules in Makefiles, a target has input and output files and a command to produce the output files. Note that in Craftr, targets can also represents *Tasks* which can be used to embed real Python functions into the build graph.

Using the *Target* class directly is usually not necessary unless you have very specific requirements and need control over the exact commands that will be executed. Or if you're just being super lazy and need the easiest script to compile a C program:

```
# craftr_module(super_lazy)
from craftr import Target, path
main = Target(
  command = 'gcc $in -o $out',
  inputs  = path.local(['src/main.c', 'src/util.c']),
  outputs = 'main'
)
```

The substition of `$in` and `$out` is conveniently done by Ninja.

```
$ craftr .main
[1/1] gcc /home/niklas/Desktop/example/src/main....til.c -o /home/niklas/Desktop/example/build/main
```

CHAPTER 6

Tasks

Tasks were initially designed as functions doing convenient operations that can be invoked from the command-line, they can also be used to export any function as a "command" to the Ninja manifest and have the production of output files implemented in Python.

A common use-case for tasks is to generate an archive from the build products to have it ready for distribution. Below you can find a simple example using the *archive* and *git* extension modules:

```
#craftr_module(myapp)
from craftr import path, task, info
from craftr.ext import archive, git, platform

git = git.Git(project_dir)
obj = platform.cc.compile(sources = path.glob('src/*.c'))
bin = platform.ld.linkn(inputs = obj, output = 'myapp')

@task(requires = [bin])
def archive():
  archive = Archive(prefix = 'myapp-{}'.format(git.describe()))
  archive.add('res')         # Add a directory to the archive
  archive.add(bin.outputs)   # Add the produced binary
  archive.save()
  info('archive saved: {!r}'.format(archive.name))
```

Note: Craftr is clever enough to run a task directly if it doesn't need any Ninja targets to be built before it can be executed. For example, the following task via `craftr .hello`

```
@task
def hello():
  info('Hello, World!')
```

See also:

Tasks invoked by Ninja are executed through the *Craftr RTS*.

CHAPTER 7

Generator Functions

Most of the time you don't want to be using *Targets* directly but instead use functions to produce them with a high-level interface. It is sometimes useful to create such a target generator function first and then use it to reduce the complexity of the build script.

The Craftr standard library provides an extensive set of functions and classes that generate targets for you, most notably the C/C++ compiler toolsets.

See also:

Since C/C++ builds are very complex and strongly vary between platforms, Craftr defines a standard interface for compiling C/C++ source files as well as the linking and archiving steps.

- *Platform Interface*
- *C/C++ Compiler Interface*
- *Linker Interface*
- *Archiver Interface*

Functions that generate targets use the `craftr.TargetBuilder` that does a lot of useful preprocessing and, as the name suggests, make building *Targets* much easier.

CHAPTER 8

Frameworks

The *craftr.Framework* is in fact just a dictionary (with an additional *name* attribute) that represents a set of options for anything build related. How the data is interpreted depends on the generator function.

Frameworks are useful for grouping build information. They were designed for C/C++ builds but may find other uses as well. For example, there might be a framework for a C++ library that specifies the include paths, preprocessor definitions, linker search path and other libraries required for the library to be used in a C++ application.

For example, the Craftfile for a header-only C++ library might look as simple as this:

```
from craftr import Framework, path
from craftr.ext.libs.some_library import some_library
my_library = Framework(
  frameworks = [some_library],
  include = [path.local('include')],
  libs = ['zip'],
)
```

As you can see in the example above, frameworks can also be nested.

Targets there were generated by helper functions (as described in the *Generator Functions* section) list up the frameworks that have been used to produce the target in the *Target.frameworks* attribute. Passing a target directly as input to another generator function will automatically inherit the frameworks of that target!

```
fw = Framework(
  include = [path.local('vendor/include'),
  libpath = [path.local('vendor/lib')],
  libs = ['vendorlib1', 'vendorlib2']
)

obj = cc.compiler(
  sources = path.glob('src/*.c'),
  frameworks = [fw]
)

bin = ld.link(
  inputs = obj
  # we don't need to specify "fw" again, it is inherited from "obj"
)
```

CHAPTER 9

Build Options

Options for the build process are entirely read from environment variables. The *craftr.options.get()* function is a convenient method to read the options from the environment.

In Craftr, options can be specified local for a module or globally for all modules. A local option is actually prefixed by the full name of the Craftr module.

```
#craftr_module(app)
from craftr import options
name = options.get('name')
debug = options.get_bool('debug')

info('Hello {}, you want a {} build?'.format(name, 'debug' if debug else 'release'))
```

The options can be specified locally using the following methods:

```
craftr -D.name="John Doe" -D.debug
craftr -Dapp.name="John Doe" -Dapp.debug
app.name="John Doe" app.debug="true" craftr   # assuming your shell supports this syntax
```

They can be set globally like this:

```
craftr -Dname="John Doe" -Ddebug
name="John Doe" debug="true" craftr   # assuming your shell supports this syntax
```

Options and environment variables can also be set from `craftrc.py` files.

Oh, and say hello to John!

```
Hello John Doe, you want a debug build?
```

CHAPTER 10

craftrc.py Files

Before anything, Craftr will execute a `craftrc.py` file if any exist. This file can exist in the current working directory and/or the users home directory. Both will be executed if both exist! You can prevent Craftr from executing these files by passing `--no-rc`. You can also tell it to execute a specific file with the `--rc` parameter (can be combined).

This file is not executed in a Craftr module context and hence should not declare any targets, but it can be used to set up environment variables and options.

For example, for using the craftr.ext.qt5 module on Windows, you could use this `craftrc.py` file in the home directory to let the Craftr Qt5 module know where the Qt5 headers and libraries are located.

```
from os import environ
if 'Qt5Path' not in environ:
  environ['Qt5Path'] = 'D:\\lib\\Qt\\5.5\\msvc2013_64'
```

Note that you can still specify a different `Qt5Path` via the command line that will override the value set in the `craftrc.py` file because the environment variables are set in the following order:

1. Variables from the parent process/shell
2. Variables prefixed on the command-line (like `VAR=VALUE craftr ...`) if your shell supports it
3. `craftrc.py` files that modify the `craftr.environ`
4. Options passed via the `-D, --define` command-line parameter
5. Craftr modules that modify the `craftr.environ`

CHAPTER 11

Colorized Output

Craftr colorizes output by default if it is attached to a TTY. If it is not but colorized output is still desired, `CRAFTR_ISATTY` can be set to `true` in the environment. Also, colorized output can be disabled by setting the variable to `false` instead. For any other value, default behaviour applies.

CHAPTER 12

Debugging

Not only can you debug your Craftr build scripts with the `pdb` module, but you can also increase the verbosity level for more verbose output. This is very useful for tracing down warnings or locations of errors in the output, eg.:

```
λ craftr --skip-build
you really shouldn't do it that way!
```

To find the location of that line, we can pass `-v`.

```
λ craftr --skip-build -v
detected ninja v1.6.0
cd "build"
load 'craftr.ext.test'
(craftr.ext.test, line 4): you really shouldn't do it that way!
exporting 'build.ninja'
```

Now if you're really having trouble finding out how the Python script actually gets there, you can enable a stacktrace with each line that is output with `-vv`.

```
λ craftr --skip-build -vv
detected ninja v1.6.0
cd "build"
load 'craftr.ext.test'
(craftr.ext.test, line 4): you really shouldn't do it that way!
  In <module> (F:\Python34\Scripts\craftr-script.py, line 9)
  In main() (c:\users\niklas\repos\craftr-build\craftr\craftr\__main__.py, line 256)
  In import_module() (f:\python34\lib\importlib\__init__.py, line 109)
  In load_module() (c:\users\niklas\repos\craftr-build\craftr\craftr\ext.py, line 245)
  In <craftr.ext.test> (Craftfile.py, line 4)
exporting 'build.ninja'
```

This output is also nicely colorized if you're in a terminal that supports ANSI color codes.

CHAPTER 13

Additional Links

- Craftr extension modules
- Projects using Craftr

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

C

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y