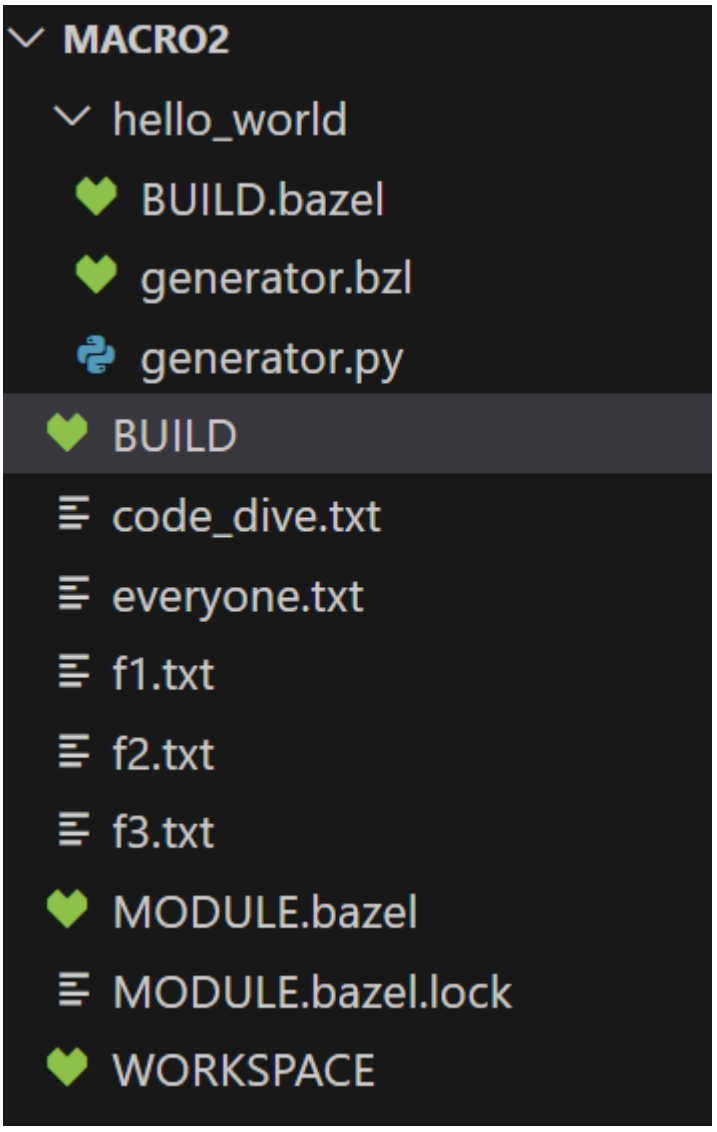
Macros & Toolchain

Macro example (takes input text file, generates cpp file and executes the cpp file)

1. **Create folder macro2**



2. **macro2/BUILD**

```
3. load("@rules_cc//cc:defs.bzl", "cc_binary")
4. load("//hello_world:generator.bzl", "hello_world")
5.
6. hello_world(
7.     name = "code_dive",
8. )
```

```
9.
10.hello_world(
11.    name = "everyone",
12.)
13.
14.cc_binary(
15.    name = "hello_world_code_dive",
16.    srcs = [":code_dive"],
17.)
18.
19.cc_binary(
20.    name = "hello_world_everyone",
21.    srcs = [":everyone"],
22.)
23.
24.
25.
```

3. macro2/hello_world/BUILD

```
load("@rules_python//python:defs.bzl", "py_binary")

py_binary(
    name = "generator",
    srcs = ["generator.py"],
    visibility = ["//visibility:public"],
)
```

4. macro2/hello_world/generator.bzl

```
def hello_world(name, visibility = None):
    native.genrule(
        name = name,
        srcs = [name + ".txt"],
        outs = [name + ".cpp"],
        cmd = "$(location //hello_world:generator) $< $@",
        tools = ["//hello_world:generator"],
```

```
        visibility = visibility,
    )
```

5. macro2/hello_world/generator.py

```
import argparse
import os

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("input_file", help="Text file containing the message
to be displayed in the hello world")
    parser.add_argument("output_file", help="Cpp file that will contain the
hello world program with the provided message")
    args = parser.parse_args()
    hello_world_message = "World"
    with open(args.input_file, 'r') as message_file:
        hello_world_message = message_file.readline()
    with open(args.output_file, 'w') as hello_world_program_file:
        hello_world_program_file.write('#include <iostream>\n')
        hello_world_program_file.write('\n')
        hello_world_program_file.write('int main()\n')
        hello_world_program_file.write('{\n')
        hello_world_program_file.write(f'    std::cout << "Hello
{hello_world_message}!" << std::endl;\n')
        hello_world_program_file.write('}\n')

if __name__ == "__main__":
    main()
```

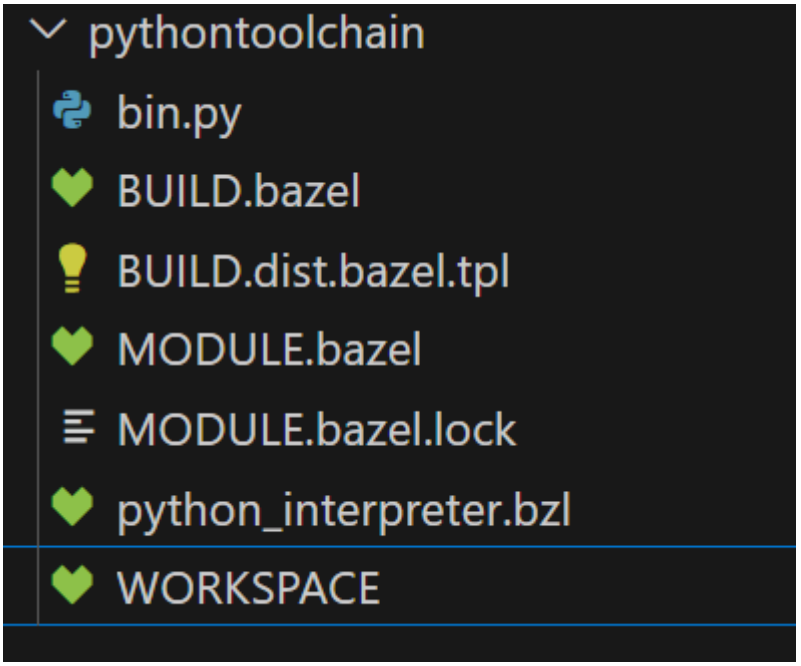
6. Navigate to macro2 and build the targets

$ bazel build //: hello_world_code_dive

$ bazel build //: hello_world_everyone

Toolchain (creating and registering python toolchain with the help of platform, constraints and rules)

1. create a folder pythontoolchain



2. bin.py

```
import sys


print(sys.version)
```

3. pythontoolchain/BUILD

```
py_binary(
    name = "bin",
    srcs = ["bin.py"],
    visibility = ["//visibility:public"],
)
```

4. BUILD.dist.bazel.tpl

```
load("@bazel_tools//tools/python:toolchain.bzl",
"py_runtime_pair")

filegroup(
    name = "files",
    srcs = glob(["install/**"], exclude = ["**/* *"]),
    visibility = ["//visibility:public"],
)

filegroup(
    name = "interpreter",
    srcs = ["{interpreter_path}"],
    visibility = ["//visibility:public"],
)

# The py_runtime target denotes a platform runtime or a hermetic
runtime.
# The platform runtime (system runtime) by its nature is
non-hermetic.
# This py_runtime target is for our hermetic Python.
py_runtime(
    name = "py_runtime",
    files = [":files"],
    interpreter = ":interpreter",
```

```
    python_version = "PY3",
    visibility = ["//visibility:public"],
)

# A py_runtime_pair is used to couple hermetic Python2 and
Python3 runtimes into a toolchain.
# We're not supporting py2, hence we pass None.
py_runtime_pair(
    name = "py_runtime_pair",
    py2_runtime = None,
    py3_runtime = ":py_runtime",
)

toolchain(
    name = "toolchain",
    exec_compatible_with = [
        {constraints},
    ],
    target_compatible_with = [
        {constraints},
    ],
    toolchain = ":py_runtime_pair",

    # We're just using the builtin Python toolchain type.
    # A toolchain_type is simply a name that describes the type of
the toolchain.
    toolchain_type = "@bazel_tools//tools/python:toolchain_type",
)
```

5. python_interpreter.bzl

```
load("@bazel_tools//tools/python:toolchain.bzl",
"py_runtime_pair")

filegroup(
    name = "files",
    srcs = glob(["install/**"], exclude = ["**/* *"]),
```

```
    visibility = ["//visibility:public"],
)

filegroup(
    name = "interpreter",
    srcs = ["{interpreter_path}"],
    visibility = ["//visibility:public"],
)

# The py_runtime target denotes a platform runtime or a hermetic
runtime.
# The platform runtime (system runtime) by its nature is
non-hermetic.
# This py_runtime target is for our hermetic Python.
py_runtime(
    name = "py_runtime",
    files = [":files"],
    interpreter = ":interpreter",
    python_version = "PY3",
    visibility = ["//visibility:public"],
)

# A py_runtime_pair is used to couple hermetic Python2 and
Python3 runtimes into a toolchain.
# We're not supporting py2, hence we pass None.
py_runtime_pair(
    name = "py_runtime_pair",
    py2_runtime = None,
    py3_runtime = ":py_runtime",
)

toolchain(
    name = "toolchain",
    exec_compatible_with = [
        {constraints},
```

```
    ],
    target_compatible_with = [
        {constraints},
    ],
    toolchain = ":py_runtime_pair",

    # We're just using the builtin Python toolchain type.
    # A toolchain_type is simply a name that describes the type of the toolchain.
    # We could define our own toolchain_type but there is no need to for this use case.
    toolchain_type = "@bazel_tools//tools/python:toolchain_type",
)
```

6. pythontoolchain/WORKSPACE

```
workspace(
    name = "rules_py_simple",
)

load("@rules_py_simple//:python_interpreter.bzl", "py_download")

py_download(
    name = "py_darwin_x86_64",
    arch = "x86_64",
    os = "darwin",
    sha256 =
"fc0d184feb6db61f410871d0660d2d560e0397c36d08b086dfe115264d1963f4",
    urls =
["https://github.com/indygreg/python-build-standalone/releases/download/20211017/cpython-3.10.0-x86_64-apple-darwin-install_only-20211017T1616.tar.gz"],
)

py_download(
    name = "py_linux_x86_64",
```

```
    arch = "x86_64",
    os = "linux",
    sha256 =
"eada875c9b39cc4bf4a055dd8f5188e99c0c90dd5deb05b6c213f49482fe20a6",
    urls =
["https://github.com/indygreg/python-build-standalone/releases/download/202110
17/cpython-3.10.0-x86_64-unknown-linux-gnu-install_only-20211017T1616.tar.gz"]
,
)

# The //:toolchain target points to the toolchain target we wrote in the BUILD
file template.
register_toolchains(
    "@py_darwin_x86_64//:toolchain",
    "@py_linux_x86_64//:toolchain",
)
```

7. navigate to pythontoolchain and run the bin target

$ bazel run //:bin