omnifig

Release 0.6.3

Felix Leeb

Oct 25, 2022

INTRO

CHAPTER

ONE

INSTALL

Everything is tested with Python 3.7 on Ubuntu 18.04 and Windows 10, but in principle it should work on any system that can handle the dependencies.

You can install this package through pip:

```
pip install omnifig
```

You can clone this repo and install the local version for development:

```
git clone https://github.com/felixludos/omni-fig
pip install -e ./omni-fig
```

CHAPTER

TWO

QUICKSTART

A project only requires a yaml file called `.fig.yml` or similar (see documentation), however, it is also suggested to create a directory called `config` to contain any config yaml files that should automatically be registered when the project is loaded. Usually, when loading a project, that requires running some python files, relative path to the top level source file to run should be specified in the project info file (`.fig.yml`) under the key `src`. Below is an example of a simple `omni-fig` project with all the suggested bells and whistles:

```
project/ - project root name (can be anything)
│
├── config/ - any yaml config files that should automatically be registered
│   ├── dir1/
│   │   ├── myconfig2.yaml
│   │   └── ... - any configs will be registered with the relative path as prefix (
↪"dir1/")
│   │
│   ├── myconfig1.yaml
│   └── debug.yaml - config to be automatically used in debug mode
│
├── src/ - any python source files
│   ├── __init__.py - python file to be called to load project as a package into␣
↪omnifig.projects
│   ├── script1.py - any additional source files
│   └── ...
│
├── .fig.yml - project info file
└── ...
```

For the example above, `.fig.yml` should contain something like:

```
name: myproject
package: src
```

To specify that `src/` contains the code neccessary load the project.

Inside the python package `src/` you can register any `Components`, `Modifiers`, `Scripts`, or configs needed for the project. For example, `src/__init__.py` might look like:

```
import omnifig as fig

@fig.Component('myconverter') # registers a new component (any class or function to␣
↪be specified in the config)
class Converter:
    def __init__(self, A): # when creating a component, the input is the config␣
↪object at the corresponding node
        self.rates = A.pull('rates', {})
```

(continues on next page)

(continued from previous page)

```
    def to_usd(self, value, currency):
        if currency in self.rates:
            return value / self.rates[currency]
        return value

@fig.AutoModifier('sketchy') # registers a new automodifier (used to dynamically␣
↪modify components)
class Sketchy:
    def __init__(self, A):
        super().__init__(A) # AutoModifiers become subclasses of the Component they␣
↪modify

        self.fudge_the_numbers = A.pull('fudge_the_numbers', True)

    def to_usd(self, value, currency):
        value = super().to_usd(value, currency)
        if self.fudge_the_numbers:
            return value * 0.9
        return value


@fig.Script('myscript', description='Does something awesome') # registers a new␣
↪script called "myscript"
def run_train_model(A): # config object containing all necessary config info
    print('Running myscript!')

    arg1 = A.pull('arg1') # gets the value corresponding to "arg1" in the config

    # pull the value corresponding to the key "arg2" starting from the node at "some.
↪deep"
    # defaults to "[default value]" if that fails
    arg2 = A.pull('some.deep.arg2', '[default value]')

    # set (and get) arg2 to "myvalue", unless it already exists
    # also this will automatically create the node "other_branch" if it doesn't␣
↪already exist
    arg3 = A.push('other_branch.arg3', 'myvalue', overwrite=False)

    # when a node (eg. "converter") contains the key "_type" (and optionally "_mod")␣
↪it is treated as a component
    A.push('converter._type', 'myconverter', overwrite=False)

    # values can be lists/dicts (even nested)

    budget, unit = A.pull('mymoney', [1000000000, 'Zimbabwe-dollars'])

    converter = A.pull('converter', None) # when pulling components, they objects are␣
↪automatically created

    if converter is not None:
        budget = converter.to_usd(budget, unit)
    else:
        raise Exception('No converter to confirm budget')

    # ... maybe do something interesting with all that money
```

(continues on next page)

(continued from previous page)

```
    msg = "I'm {}a millionaire".format('' if budget > 1e6 else 'not ')
    print(msg)

    return msg # anything this script should return
```

Any function or class that should be specified in the config should be registered as a `Component`. When "pulling" a component (a config node that contains the "_type" key), the config system will automatically get the corresponding class/function and run it (returning the created instance/output). You can also define and register `Modifiers` to dynamically specify modifications that you want to make to the components in the config (using the "_mod" key in the same node as "_type").

It is highly recommended that you create a profile info yaml file and set the environment variable `FIG_PROFILE` to the full path to that profile info file. For example, the profile might contain:

```
name: mycomputer

projects:
    myproject: /path/to/myproject # path to the "myproject" directory mentioned above
```

As you create new projects, you can add those to the profile info file so they can loaded from anywhere. By default, only the project in the current working direcory is loaded (and any "related" projects thereof), however that can also be changed in the profile info file (see the documentation).

With this setup, you should be able to run all of the below (from the terminal inside `myproject/`):

```
# execute myscript without any config files or arguments
fig myscript

# execute myscript in debug mode ("-d") and with config file "dir1/myconfig2"
fig -d myscript dir1/myconfig2

# execute myscript with "myconfig1" as config updated by command line argument
fig myscript myconfig1 --arg1 cmdline

# execute myscript with merged config file and command line arguments
python script1.py myconfig1 dir1/myconfig2 --some.deep.arg2 10.2

# execute myscript in debug mode with merged config and command line argument
python script1.py -d myconfig1 dir1/myconfig2 --converter._mod.sketchy 1 --arg1
```

It might be worth taking a look at the resulting config object looks like for each of these commands (and depending on what information is saved in the corresponding config files in `myproject/config/`. Note that you can use `-d` to switch to debug mode (see documentation for more info).

You might also load and run scripts in this project from a jupyter notebook (or a python console) using:

```
import omnifig as fig

fig.initialize('myproject') # load profile and project

A = fig.get_config('dir1/myconfig2', 'config1') # positional arguments can be names␣
↪of registered config files
out1 = fig.run('myscript', A)

B = fig.get_config('config1', arg1=[1,2,3]) # keyword arguments are much like command␣
↪line arguments
out2 = fig.run('myscript', B, debug=True) # meta arguments (such as "debug") can be␣
↪set using keyword args in run()
```

(continues on next page)

(continued from previous page)

```
C = fig.get_config(arg1='something', arg2='another thing')
C.update(B)
C.push('arg1', 'something else') # the config object can be modified with push()/
↪update()
out3 = fig.run('myscript', C)

# quick_run effectively combines get_config and
out4 = fig.quick_run('myscript', 'config1', use_gpu=True)
```

While this example should give you a basic idea for what a project might look like, this only touches on the basics of what you can do with `omni-fig`. I strongly recommend you check out the documentation. for more information, additionally there are some examples of real projects that use `omni-fig` such as omnilearn and No-Nonsense-News .

2.1 Top-level interface

By default, these functions provide all the basic and most important features of this package. However, they usually delegate all the real heavy-lifting to the "current" project or profile, so for more fine-grain control over exactly how your code gets organized and additional customization, take a look at the full documentation (particularly the sections about profiles and projects).

get_current_project()
Get the current project, assuming a profile is loaded, otherwise returns None

get_project(*ident=None*)
Checks the profile to return (and possibly load) a project given the name or path `ident`

entry(*script_name=None*)
Recommended entry point when running a script from the terminal. This is also the entry point for the `fig` command.

This collects the command line arguments in `sys.argv` and overrides the given script with `script_name` if it is provided

Parameters **script_name** – script to be run (may be set with arguments) (overrides other arguments if provided)

Returns None

main(**argv*, *script_name=None*)
Runs the desired script using the provided `argv` which are treated as command line arguments

Before running the script, this function initializes `omni-fig` using *`initialize()`*, and then cleans up after running using *`cleanup()`*.

Parameters

- **argv** – raw arguments as if passed in through the terminal
- **script_name** – name of registered script to be run (may be set with arguments) (overrides other arguments if provided)

Returns output of script that is run

run(*script_name*, *config*, ***meta*)
Runs the specified script registered with `script_name` using the current project.

Parameters

- **script_name** – must be registered in the current project or defaults to the profile
- **config** – config object passed to the script
- **meta** – any meta rules that modify the way the script is run

Returns output of the script, raises MissingScriptError if the script is not found

quick_run(*script_name*, **parents*, ***args*)

Convenience function to run a simple script without a given config object, instead the config is entirely created using the provided `parents` and `args`.

Parameters

- **script_name** – name of registered script that is to be run
- **parents** – any names of registered configs to load
- **args** – any additional arguments to be provided manually

Returns script output

initialize(**projects*, ***overrides*)

Initializes omni-fig by running the "princeps" file (if one exists), loading the profile, and any active projects. Additionally loads the project in the current working directory (by default).

Generally, this function should be run before running any scripts, as it should register all necessary scripts, components, and configs when loading a project. It is automatically called when running the *`main()`* function (ie. running through the terminal). However, when starting scripts from other environments (such as in a jupyter notebook), this should be called manually after importing `omnifig`.

Parameters

- **projects** – additional projects that should be initialized
- **overrides** – settings to be checked before defaulting to `os.environ` or global settings

Returns None

cleanup(***overrides*)

Cleans up the projects and profile, which by default just updates the project/profile info yaml file if new information was added to the project/profile.

Generally, this should be run after running any desired scripts.

Parameters **overrides** – settings to check before defaulting to global settings or `os.environ`

Returns None

get_config(**contents*, ***parameters*)

Process the provided info using the current project into a config object. :param contents: usually a list of parent configs to be merged :param parameters: any manual parameters to include in the config object :return: config object

create_component(*config*)

Create a component using the current project :param config: Must contain a "_type" parameter with the name of a registered component :return: the created component

quick_create(*_type*, **parents*, ***parameters*)

Creates a component without an explicit config object. Effectively combines *get_config()* and *create_component()* :param _type: :param parents: :param parameters: :return:

register_script(*name*, *fn*, *description=None*, *use_config=False*)

Manually register a new script to the current project

register_component(*name*, *fn*, *description=None*)
Manually register a new component to the current project

register_modifier(*name*, *fn*, *description=None*, *expects_config=False*)
Manually register a new modifier to the current project

register_config(*name*, *path*)
Manually register a new config file to the current project

register_config_dir(*path*, *recursive=True*, *prefix=None*, *joiner='/'*)
Manually register a new config directory to the current project

has_script(*name*)

find_script(*name*)

view_scripts()

has_component(*name*)

find_component(*name*)

view_components()

has_modifier(*name*)

find_modifier(*name*)

view_modifiers()

has_config(*name*)

find_config(*name*)

view_configs()

CHAPTER

THREE

PHILOSOPHY

Python is an incredibly versatile language. The dynamic nature and expansive community allows developers to program with virtually no overhead, developing anything from highly specialized applications that make use of a plethora of packages to general scripts that fit into 100 lines of code.

However, with great power comes great responsibility: in this case that means keeping our many little scripts and packages organized (and ideally documented and with unit tests). There are already some excellent packages that take care of documentation `sphinx` (with `readthedocs`) and a very simple testing framework `pytest` (with `Travis CI`). These tools can ensure the understandability and functionality of our code, but what about keeping the code itself organized?

How can we minimize code duplication while still being able to easily change or add new functionality or run everything in a variety of different execution environments? This is the purpose of *omni-fig*. While there are a variety of organizational tools included in *omni-fig* (such as machine profiles, projects, run modes with customizable meta arguments), the most important components are the registration and the config system. The registration system allows for a fine-grained control to select which code is run and how. Meanwhile the config system keeps all the necessary arguments and parameters organized in an intuitive hierarchical structure enabling easy modification of what the scripts actually do.

3.1 Registration: Beyond `import`

Python's native `import` system is rather convenient (and significantly nicer than some other languages), nevertheless, for highly dynamic projects, it can slow down productivity to constantly make sure all the right code is made available where it is needed.

The registration system in `omni-fig` offers a much more fine-grained alternative (much of which is built directly into the config system). The idea is to register different pieces of code as a *Script*, *Component*, or *Modifier* using the corresponding decorators depending on how it is meant to be used. Anything registered as a *Script* (usually a function) can be run in a variety of ways (see "run modes" below) but essentially act as a self contained operation. A *Component* is some piece of code (usually a class) that builds an arbitrarily complex object to be used later (such as in a *Script*). Finally, *Modifiers* allow *Components* to be modified dynamically (however, *Modifiers* are a bit more involved to understand, see below for more info and examples). The most important distinction between *Scripts* and *Components* is that *Components* are created automatically by the config system, while *Scripts* have to be called/executed manually and that *Components* can be modified with *Modifiers* (for more details see the corresponding sections below).

Once registered, *Scripts*, *Components*, and *Modifiers* can be used anywhere mitigating the need for lots of `import` statements in every new file. Additionally, *Scripts* can be run using all of the registered "run modes" (eg. run from the terminal, with a debugger, etc.).

Another major benefit of using a registration system is that the registered objects can be referred to using their registered names (which are strings instead of python classes/objects). This allows config files to explicitly specify complex objects that can be built dynamically (see Config System for more info).

3.2 Config System

The code you write is only as valuable as you are able to use it in the way you want. This means, good code organization necessitates the power to specify exactly what the code should do in the form of arguments and configs. To that end, *omni-fig* provides a flexible config structure that uses a tree-like hierarchy to dynamically provide arguments for all components and subcomponents.

The hierarchical structure not only allows grouping arguments but it also allows for argument "scopes" - ie. when an argument is not found in the current node, it defaults to check the parent. More universal arguments can be set on a higher level of the tree, but then optionally be overridden in subcomponents without affecting other components.

CHAPTER

FOUR

REGISTRY

At the heart of good code organization is a flexible yet powerful way to add new features or functionality to a past, current, and even future projects. `omni-fig` accomplishes this by relying on registries to manage all of the most important pieces of code that may be explicitly addressed/referred to in the config.

The first registry registers all scripts which are essentially the top level interface for how a user is to interact with the code (by calling scripts). These scripts can be called from the terminal (see *Running Scripts* for more details and examples) or executed in environments like jupyter notebooks or an IDE debugger (eg. Pycharm).

Next, is the component registry. A component is any atomic piece of code that might be specified in the config. As the config object is a yaml file, there are no python classes or objects there in (aside from dicts, lists and primitives). Instead, if the user wants to use a user defined class or function, it can be registered as a component, and then the component can be referred to in the config with the key `_type`.

Finally, modifiers can be registered and used to wrap existing components to further customize the behavior of components dynamically from the config. There are two special kinds of modifiers: `AutoModifier()` and `Modification()`. An `AutoModifier()` essentially acts as a child class of whatever component it is wrapping. Meanwhile a `Modification()` is used to wrap or modify a component after it has been created. The example below (and elsewhere) will hopefully help elucidate how modifiers can be used with components.

For an extended example in how all three registries might be used for a simple project where we sample from a guassian distribution and then record the particularly low probability events, which might be implemented and registered like so:

```
import sys
import random
import omnifig as fig

@fig.Script('sample-low-prob')
def sample_low_prob(A): # config object

    mylogger = A.pull('logger') # create a logger object according to specifications␣
↪in the config

    num_events = A.pull('num_events', 10) # default value is 10 if "num_events" is␣
↪not specified in the config

    interest_criterion = A.pull('criterion', 5.)
    important_criterion = A.pull('important_criterion', 5.2)

    mu, sigma = A.pull('mu',0.), A.pull('sigma', 1.)
    sigma = max(sigma, 1e-8) # ensure sigma is positive

    print('Sampling...')
```

(continues on next page)

(continued from previous page)

```
    events = []
    count = 0
    while len(events) < num_events:
        x = random.gauss(mu, sigma)
        if abs(x) > interest_criterion:
            mylogger.log_line(f'Found important {x:.2f}\n', important=abs(x)>
↪important_criterion)
            events.append(x)
        count += 1

    mylogger.log_line(f'Finding {num_events} low prob samples required {count}␣
↪samples.\n',
                      include_credits=True, important=True)

    mylogger.close()

    return events
```

In this example project, we may require a logger (called `logger` above) to print information to `stdout` or a file, and we can register components to implement the different choices and corresponding arguments.

```
@fig.AutoComponent('stdout') # automatically pulls all arguments in signature before␣
↪creating
def _get_stdout(): # in this case, we don't need any arguments
    return sys.stdout

@fig.AutoComponent('file')
def _get_file(path):
    return open(path, 'a+')

@fig.Component('mylogger')
class Logger:
    def __init__(self, A): # "A" refers to the config object
        self.always_log = A.pull('always_log', False) # value defaults to False if␣
↪not found in the config
        self.print_stream = A.pull('print_stream', None) # values can also be␣
↪components themselves
        self.credits = A.pull('credits', []) # pulled values can also be dicts or␣
↪lists (with defaults)
        if not isinstance(self.credits, list):
            self.credits = list(self.credits)

    def log_line(self, line, stream=None, important=False, include_credits=False):
        if stream is None:
            stream = self.print_stream
        if stream is not None and (important or self.always_log):
            stream.write(line)
            if include_credits and len(self.credits):
                stream.write('Credits: {}\n'.format(', '.join(self.credits)))

    def close(self):
        if self.print_stream is not None:
            self.print_stream.close()
```

This example shows how `Component()` and `AutoComponent()` may be used with both classes and functions. The config (eg. registered as `myconfig1`) may contain something like:

```yaml
num_events: 5

logger:
  _type: mylogger
  credits: [Gauss, Hamilton, Fourier]
  print_stream._type: stdout    # "." is treated like a sub-dict
```

Or (say, `myconfig2`):

```yaml
always_log: True  # as this argument is in a parent dict of "logger" it will still be␣
↪found within "logger".
logger:
  _type: mylogger
  print_stream:
    _type: file
    path: 'log_file.txt'    # "." is treated like a sub-dict
```

Additionally, components can be modified in the config using `Modifier()`, `AutoModifier()`, and `Modification()`. Modifiers essentially act as additional decorators that can dynamically be specified in the config to change the bahavior of components before (eg. `Modifier()` or `AutoModifier()`) or after (`Modification()`) creating the component.

To add on to our previous example:

```python
@fig.AutoModifier('multi')
class MultiStream:
    def __init__(self, A):

        streams = A.pull('print_streams', '<>print_stream', []) # use prefix "<>" to␣
↪default to a different key
        if not isinstance(streams, (list, tuple)):
            streams = [streams]

        A.push('print_stream', None) # push to replace values in the config

        super().__init__(A) # initialize any dynamically added superclasses (->␣
↪Logger)

        self.print_streams = streams

    def log_line(self, line, stream=None, important=False, include_credits=False):

        if stream is not None:
            return super().log_line(line, stream=stream, important=important, include_
↪credits=include_credits)
        for stream in self.print_streams:
            return super().log_line(line, stream=stream, important=important, include_
↪credits=include_credits)

    def close(self):
        for stream in self.print_streams:
            stream.close()

@fig.Modification('remove-credits')
def remove_names(logger, A):
    for name in A.pull('remove_names', []):
        if name in logger.credits:
```

(continues on next page)

(continued from previous page)

```
            logger.credits.remove(name)
            print(f'Removed {name} from credits')
    return logger
```

And some associated configs might include (`config3`):

```
parents: [config1] # all these registered configs will be loaded and merged with this one

path: 'backup-log.txt'

logger:
  _mod: multi

  print_streams:
  - _type: file
  - _type: stdout
```

Or, finally (`config4`):

```
parents: [config2, config3]

remove_names: [Fourier]

logger._mod: [multi, remove-credits]
```

Now, if your head isn't spinning from the complicated merging and defaulting of configs, then perhaps you can figure out what path we will actually end up using as our log file when using `config4`?

The answer is `backup-log.txt` because the `AutoModifier() multi` starts from the `logger.print_streams` branch, which does not get merged with the `logger.print_stream` branch (which contains `path : 'log_file.txt'`), so when defaulting towards the root, `log_file.txt` is not encountered. For more information, the code for this example can be found in `examples/gauss_fun`.

Another part of this example that warrants careful consideration is how the `AutoModifier() multi` is used. The trick is that an `AutoModifier()` actually dynamically creates a new child class of the registered `AutoModifier()` type and the original type of the component (for that reason `AutoModifier()` must be a class, not a function, and they only work on components that are classes). In this case, the dynamically created type will be called `MultiStream_Logger` with the method resolution order (MRO) `[MultiStream, Logger, object]`.

Note that the `AutoModifier()` can be paired with, in principle, any component (although some will raise errors), which effectively means an `AutoModifier()` allows changing the behavior of any component, even ones that haven't even been written yet. While `AutoModifier()` is one of the most powerful features of the registry system in `omni-fig`, they are consequently also rather advanced, so particular care must be taken when using them.

Script (*name*, *description=None*, *use_config=True*)

Decorator to register a script

Parameters

- **name** – name of script
- **description** – a short description of what the script does
- **use_config** – `True` if the config should be passed as only arg when calling the script function, otherise it will automatically pull all arguments in the script function signature

Returns decorator function expecting a callable

AutoScript(*name*, *description=None*)
Convienence decorator to register scripts that automatically extract relevant arguments from the config object

Parameters

- **name** – name of the script
- **description** – a short description of what the script does

Returns decorator function expecting a callable that does not expect the config as argument (otherwise use *Script()*)

Component(*name=None*)
Decorator to register a component

NOTE: components should usually be types/classes to allow modifications

Parameters **name** – if not provided, will use the __name__ attribute.

Returns decorator function

AutoComponent(*name=None*, *aliases=None*, *auto_name=True*)
Instead of directly passing the config to an AutoComponent, the necessary args are auto filled and passed in. This means AutoComponents are somewhat limited in that they cannot modify the config object and they cannot be modified with AutoModifiers.

Note: AutoComponents are usually components that are created with functions (rather than classes) since they can't be automodified. When registering classes as components, you should probably use *Component* instead, and pull from the config directly.

Parameters

- **name** – name to use when registering the auto component
- **aliases** – optional aliases for arguments used when autofilling (should be a dict[name,list[aliases]])

Returns decorator function

Modifier(*name=None*, *expects_config=False*)
Decorator to register a modifier

NOTE: a *Modifier* is usually not a type/class, but rather a function (except `AutoModifiers`, see below)

Parameters

- **name** – if not provided, will use the __name__ attribute.
- **expects_config** – True iff this modifier expects to be given the config as second arg

Returns decorator function

AutoModifier(*name=None*)
Can be used to automatically register modifiers that combine types

To keep component creation as clean as possible, modifier types should allow arguments to their __init__ (other than the Config object) and only call pull on arguments not provided, that way child classes of the modifier types can specify defaults for the modifications without calling pull() multiple times on the same arg.

Note: in a way, this converts components to modifiers (but think before using). This turns the modified component into a child class of this modifier and its previous type.

In short, Modifiers are used for wrapping of components, AutoModifiers are used for subclassing components

Parameters **name** – if not provided, will use the __name__ attribute.

Returns decorator to decorate a class

Modification(*name=None*)

A kind of Modifier that modifies the component after it is created, and then returns the modified component

expects a callable with input (component, config)

Modifications should almost always be applied after all other modifiers, so they should appear at the end of _mod list

Parameters **name** – name to register

Returns a decorator expecting the modification function

CHAPTER

FIVE

CONFIG SYSTEM

The original motivation for this package was to design a system that would read arguments from files, the terminal, or directly as python objects (eg. in a jupyter notebook) and would structure so that the arguments are always used in the code where they need to be. This is accomplished chiefly by a hierarchical structure in the config much like a tree where each branch corresponds to the a dict or list of arguments (or other branches).

When reading (aka *pulling*) arguments from the config, if an argument is not found in the current branch, it will automatically defer to the higher branches (aka *parent* branch) as well, which allows users to define more or less "global" arguments depending on which node actually contains the argument.

It should be noted that despite the hierarchical structure of the config object, it can ultimately always be exported into a simple yaml file - so it should not contain any values that are primitives (`str`, `bool`, `int`, `float`, `None`) aside from the branches that behave either as a `dict` or `list`. Using the registries, the config can implicitly contain specifications for arbitrarily complex components such as objects and functions (see *Registry* for more details and an example).

Probably, the next most important feature of the config object is that a config object can be updated with other configs, and thereby "inherit" arguments from other config files. This inheritance behaves analogously to python's class inheritance in that each config can have arbitrarily many parents and the full inheritance tree is linearized using the "C3" linearization algorithm (so no cycles are permitted). The configs are updated in reverse order of precedence (so that the higher precedence config file can override arguments in the lower precedence files).

Generally, arguments are read from the config object using `pull()` and individually updated or set using `push()`.Both `pull()` and `push()` supports *deep* gets and sets, which means you can get and set arguments arbitrarily deep in the config hierarchy using "." to "dereference" (aka "go into") a branch. When pulling, additional default values can be provided to process if the key is not found. This is especially useful in conjunction with another feature called aliasing, where arguments can reference each other (see the example below).

For example, given the config object that is loaded from the this yaml file (registered as `myconfig`):

```
favorites:
    games: [Innovation, Triumph and Tragedy, Inis, Nations]
    language: Python
    activity: <>games.0

wallpaper:
    color: red

jacket:
    size: 30

nights: 2
trip:
    - location: London
      nights: 3
```

(continues on next page)

(continued from previous page)

```
    - location: Berlin
    - location: Moscow
      nights: 4

app:
    price: 1.99
    color: <>wallpaper.color
```

When this yaml file is loaded (eg. `config = omnifig.get_config('myconfig')`), we could use it like so:

```
assert config.pull('favorites.language') == 'Python'
assert config.pull('favorites.0') == 'Innvoation'
assert config.pull('app.color') == 'red'
assert config.pull('app.publisher', 'unknown') == 'unknown'
assert config.pull('jacket.color', '<>wallpaper.color') == 'red'
assert config.pull('favorites.activity') == 'Innovation'
assert config.pull('jacket.price', '<>price', '<>total_cost', 'too much') == 'too much
↪'
assert config.pull('trip.0.location') == 'London'
assert config.pull('trip.1.nights', 4) == 2
```

While this example should give you a sense for what kind of features the config system offers, a comprehensive list of features follows.

5.1 Keys

In addition to the behavior described above, the keys (or indices) in a config branch have the following features (where `{}` refers to any value):

- `push()`/`pull()` `'_{}'` - protected argument not visible to child branches when they defer to parents
- `push()`/`pull()` `'__{}'` - volatile argument is not exported when saving config to yaml (can be used for non-yamlifiable data)
- `push()`/`pull()` `({1},{2}, ...)` - *deep* key `[{1}][{2}]`
- `push()`/`pull()` `'{1}.{2}'` - *deep* key as str `['{1}']['{2}']`
- `push()`/`pull()` `'{1}.{2}'` - *deep* key through list `['{1}'][{2}]` (where `{2}` is an int and `self['{1}']` is a list)
- `push()` `'{1}.{2}'` where `'{1}'` is missing - *deep* push automatically creates a new branch `'{1}'` in config and then pushes `'{2}'` to that new branch

5.2 Values

The values of arguments also have a few special features worth noting:

- `'<>{}'` - local alias use value of key `{}` starting search for the key here
- `'<o>{}'` - origin alias use value of key `{}` starting search for the key at origin (this only makes a difference when chaining aliases, origin refers to the branch where `pull()` was called)
- `_x_` - remove key if encountered (during update) remove corresponding key it it appears in the config being updated

- `__x__` - cut deferring chain of key behave as though this key didn't exist (and don't defer to parent)

5.3 Code

class ConfigType(*parent=None*, *printer=None*, *prefix=None*, *safe_mode=False*, *project=None*, *data=None*)

Bases: `omnibelt.transactions.Transactionable`

The abstract super class of config objects.

The most important methods:

- `push()` - set a parameter in the config
- `pull()` - get a parameter in the config
- `sub()` - get a sub branch of this config
- `seq()` - iterate through all contents
- `update()` - update a config with a different config
- `export()` - save the config object as a yaml file

Another important property of the config object is that it acts as a tree where each node can hold parameters. If a parameter cannot be found at one node, it will search up the tree for the parameter.

Config objects also allow for "deep" gets/sets, which means you can get and set parameters not just in the current node, but any number of nodes deeper in the tree by passing a list/tuple of keys or keys separated by "." (hereafter called the "address" of the parameter).

Note: that all parameters must not contain "." and should generally be valid python identifiers (strings with no white space that don't start with a number).

__deepcopy__(*memodict={}*)

__copy__()

copy()
shallow copy of the config object

pythonize()

classmethod convert(*data*, *recurse*)
used by configurize to turn a nested python object into a config object

sub(*item*)
Used to get a subbranch of the overall config :param item: address of the branch to return :return: config object at the address

update(*other*)
Used to merge two config nodes (and their children) together

This method must be implemented by child classes depending on how the contents of the node is stored

Parameters **other** – config node to overwrite `self` with

Returns None

pull(*item*, **defaults*, *silent=False*, *ref=False*, *no_parent=False*, *as_iter=False*, *raw=False*)
Top-level function to get parameters from the config object (including automatically creating components)

Parameters

- **item** – address of the parameter to get
- **defaults** – default values to use if `item` is not found
- **silent** – suppress printing message that this parameter was pulled
- **ref** – if the parameter is a component that has already been created, get a reference to the created component instead of creating a new instance
- **no_parent** – don't default to a parent node if the `item` is not found here
- **as_iter** – return an iterator over the selected value (only works if the value is a dict/list)

Returns processed value of the parameter (or default if `item` is not found, or raises a `MissingConfigError` if not found)

pull_self(*name=None*, *silent=False*, *as_iter=False*, *raw=False*, *ref=False*)
Process self as a value being pulled.

Parameters

- **name** – Name given to self for printed message
- **silent** – suppress printing message
- **as_iter** – Return self as an iterator (has same effect as calling `seq()`)

Returns the processed value of self

push(*key*, *val*, **_skip*, *silent=False*, *overwrite=True*, *no_parent=True*, *force_root=False*, *process=True*)
Set `key` with `val` in the config object, but pulls `key` first so that *val* is only set if it is not found or `overwrite` is set to `True`. It will return the current value of `key` after possibly setting with `val`.

Parameters

- **key** – key to check/set (can be list or '.' separated string)
- **val** – data to possibly write into the config object
- **_skip** – soak up all other positional arguments to make sure the remaining are keyword only
- **silent** – Do not print messages
- **overwrite** – If key is already set, overwrite with (configurized) 'val'
- **no_parent** – Do not check parent object if not found in self
- **force_root** – Push key to the root config object

Returns current val of key (updated if written)

export(*path=None*)
Convert all data to raw data (using dict/list) and save as yaml file to `path` if provided. Also returns yamlified data.

Parameters **path** – path to save data in this config (data is not saved to disk if not provided)

Returns raw "yamlified" data

seq()
Returns an iterator over the contents of this config object where elements are lazily processed during iteration (see `ConfigIter` for details).

Returns iterator over all arguments in self

replace_vals(*replacements*)

set_silent(*silent=True*)
Sets whether pushes and pulls on this config object should be printed out to stdout

silence(*silent=True*)

silenced(*setting=True*)
Returns a context manager to silence this config object

is_root()
Check if this config object has a parent for defaults

set_parent(*parent*)
Sets the parent config object to be checked when a parameter is not found in *self*

get_parent()
Get parent (returns None if this is the root)

set_process_id(*name=None*)
Set the unique ID to include when printing out pulls from this object

get_root()
Gets the root config object (returns `self` if `self` is the root)

_set_prefix(*prefix*)

_reset_prefix()

_get_hidden_prefix()

set_project(*project*)

get_project()

set_safe_mode(*safe_mode*)

in_safe_mode()

purge_volatile()
Recursively remove any items where the key starts with "__"

Must be implemented by the child class

Returns None

_single_set(*key*, *val*)

_missing_key(*key*, **context*)

class ConfigDict(*parent=None*, *printer=None*, *prefix=None*, *safe_mode=False*, *project=None*, *data=None*)
Bases: *omnifig.config.ConfigType*, `omnibelt.basic_containers.tdict`

Dict like node in the config.

Keys should all be valid python attributes (strings with no whitespace, and not starting with a number).

NOTE: avoid setting keys that start with more than one underscore (especially '__obj') (unless you really know what you are doing)

__deepcopy__(*memodict={}*)

__copy__()

copy()

replace_vals(*replacements*)

classmethod convert(*data*, *recurse*)

update(*other*)
Merge self with another dict-like object

Parameters **other** – must be dict like

Returns None

purge_volatile()
Recursively remove any items where the key starts with "__"

Returns None

class EmptyElement
Bases: object

class ConfigList(**args*, *empty_fill_value=<class 'omnifig.config.EmptyElement'>*, ***kwargs*)
Bases: *omnifig.config.ConfigType*, omnibelt.basic_containers.tlist

List like node in the config.

__deepcopy__(*memodict={}*)

__copy__()

replace_vals(*replacements*)

purge_volatile()
Recursively remove any items where the key starts with "__"

Returns None

_str_to_int(*item*)
Convert the input items to indices of the list

update(*other*)
Overwrite self with the provided list other

_single_set(*key*, *val*)

push(*first*, **rest*, *overwrite=True*, ***kwargs*)
When pushing to a list, if you don't provide an index, the value is automatically pushed to the end of the list

Parameters

- **first** – if no additional args are provided in *rest*, then this is used as the value and the key is the end of the list, otherwise this is used as key and the first element in *rest* is the value
- **rest** – optional second argument to specify the key, rather than defaulting to the end of the list
- **kwargs** – same keyword args as for the ConfigDict

Returns same as for ConfigDict

append(*item*)

extend(*item*)

class ConfigIter(*origin*, *elements=None*, *auto_pull=True*, *include_key=None*, *reversed=False*)
Bases: object

Iterate through a list of parameters, processing each item lazily, ie. only when it is iterated over (with next())

view()
Returns the next object without processing the item, may throw a StopIteration exception

step()

set_auto_pull(*auto=True*)

set_reversed(*reversed=True*)

has_next()

__iter__()

configurize_nones(*s*, *recurse*)
Turns strings into None, if they match the expected patterns

CHAPTER

SIX

RUNNING SCRIPTS

There are several ways to run scripts that are registered using `omni-fig`. The most common way is through the command line using the `fig` command or from a different environment (eg. a jupyter notebook) using `run()` or `quick_run()` (depending on if the config has already been loaded or not).

The `fig` command should be used:

```
fig [-<meta>] <script> [<configs>...] [--<args>]
```

- `script` - refers to the registered name of the script that should be run. This can be "_" to specify that the script is already specified in the config. When using the `fig` command, the script is a required argument.
- `meta` - any meta rules that should be activated should use their respective (usually single letter) codes and must always be preceeded by a `-`.
- `configs` - is an ordered list of names of registered config yaml files that should be loaded and merged when creating the config object.
- `args` - any manually provided arguments to be added to the config object after loading/merging `configs`. Here each argument key must be preceded by a `--` and optionally followed by a value (which is parsed as yaml syntax), if not value is provided the key is set to `True`.

Another slightly more advanced way to run specific scripts from the terminal is by directly calling a python file that explicitly calls `entry()` or `main()`. Below is an example of the recommended way to do so:

Contents of python file called `main.py`:

```
import omnifig as fig

if __name__ == '__main__':
    fig.entry('myscript')
```

Here `myscript` must be the name of a registered script (for example when loading the associated project), or it can be left out (in which case calling the python file behaves identically to calling the `fig` command).

Now the example command from the terminal to execute `myscript` using the meta argument `d` (which switches to debug mode by default), a few registered configs (`myconfig1` and `myconfig1`), and some manually provided arguments (`myflag` and `myvalue`):

```
python main.py -d myconfig1 myconfig2 --myflag --myvalue 1234
```

Note that if a script name is provided manually (as in `main.py` in this example), then no script name (or _) must be specified from the command line.

The equivalent command without using `main.py` is:

```
fig -d myscript myconfig1 myconfig2 --myflag --myvalue 1234
```

CHAPTER

SEVEN

INITIALIZATION

It is strongly encouraged to *initialize* omni-fig before running any scripts. When calling a script from the terminal using the `fig` command, omni-fig will initialize automatically, but when running in a separate environment (such as in a jupyter notebook), it is suggested to call `omnifig.initialize()` after importing the package. More info about `initialize()` can be found in *Running Scripts*.

The initialization process:

1. runs the initial "princeps" file (if one is specified)
2. loads the profile (if one exists)
3. loads any active projects listed in the profile
4. recursively loads all related projects
5. loads the profile in the current working directory (if there is one and it hasn't been loaded yet)

7.1 Princeps File

The *princeps* file is an optional startup file that can be run to change global settings in `omni-fig`, to change the profile type, to register new project types, or to make other advanced customizations in how `omni-fig` runs.

The *princeps* file is specified by defining an environment variable `FIG_PRINCEPS_PATH` which contains the absolute path to a python file.

For most use cases it should not be necessary to specify a *princeps* file, and running one can also be disabled all together - so it is generally discouraged to use a princeps file unless absolutely necessary.

7.2 Profiles

Generally, every OS image or file system should use it's own *profile*. The profile is specified by defining an environment variable `FIG_PROFILE` which contains the absolute path to a yaml file.

While using a profile is completely optional, it is highly recommended as the profile is the primary way to specify the location of all of your projects that you may want to load. Additionally, the profile can be used to change the global settings of `omni-fig` to tailor the behavior to the specific machine/OS. Since the profile is meant to act globally for the whole file system, all paths should be absolute.

The most important contents of the profile file (all of which are optional):

- `projects` - dictionary from project name to absolute path to the project directory
- `active_project` - list of names of projects that should automatically be loaded (the paths to these projects must be in the `projects` table

- `config_paths` - list of absolute paths to config (yaml) files or directories that should be registered when loading
- `src_paths` - list of absolute paths to python files that should be run when loading
- `default_ptype` - name of the registered project type that should be used to load projects (default: `default`)
- `global_settings` - dictionary of setting names and values to be set during loading
- `autoload_local` - bool to specify whether the project in the current working directory should be loaded (default: `True`)

class Profile(***kwargs*)

Bases: *omnifig.organization.Workspace*

Generally all paths that the Profile deals with should be absolute paths as the profile operates system wide

__init__(***kwargs*)

_process(*raw*)

Processes the data from a yaml file and saves it to `self` :param raw: data from a yaml file :return: None

static is_valid_project_path(*path*)

Check if a path points to a project directory

initialize(*loc=None*)

Steps to execute during initialization including: updating global settings, loading configs, running any specified source files, loading active projects, and possibly the local project.

Parameters **loc** – absolute path to current working directory (default comes from `os.getcwd()`)

Returns None

update_global_settings()

Updates global settings with items in `self.global_settings`

load_active_projects(*load_related=True*)

Load active projects in `self.active_projects` in order, and potentially all related projects as well

Parameters

- **load_related** – load directly related projects
- **all_related** – recursively load all related projects

Returns None

get_project(*ident=None*, *load_related=True*, *all_related=False*)

Gets project if already loaded, otherwise tries to find the project path, and then loads the missing project

Parameters

- **ident** – project name or path
- **load_related** – also load related projects (before this one) (dont recursively load related)
- **all_related** – recursively load all related projects (trumps `load_related`)

Returns project object

get_project_type(*name*)

Gets the project type registered with the given name

load_project(*ident*, *load_related=True*, *all_related=False*)

Parameters

- **ident** – path or name of project (if name, then it must be profile.projects)
- **load_related** – also load related projects (before this one) (dont recursively load related)
- **all_related** – recursively load all related projects (trumps `load_related`)

Returns loaded project (or raises `NoValidProjectError` if none is found)

resolve_project_path(*ident*)
Map/complete/fix given identifier to the project path

Parameters **ident** – name or path to project

Returns correct project path or None

clear_loaded_projects()
Clear all loaded projects from memory (this does not affect registered components or configs)

contains_project(*ident*)

track_project_info(*name*, *path*)
Add project info to projects table `self.projects`

track_project(*project*)
Add project to projects table `self.projects`

is_tracked(*project*)
Check if a project is contained in projects table `self.projects`

include_project(*project*, *track=True*)
Include a project instance in loaded projects table managed by this project and optionally track a project persistently.

add_active_project(*project*)
Add a project to the list of active projects

get_active_projects()
Get a list of all projects specified as "active"

set_current_project(*project=None*)
Set the current project

get_current_project()
Get the current project (usually loaded last and local),

find_artifact(*atype*, *name*)
Search for a registered artifact from the name either in a loaded project or in the profile (global) registries

Parameters

- **atype** – component, modifier, script, or config
- **name** – registered artifact name, can use prefix to specify project (separated with ":")

Returns artifact entry (namedtuple)

has_artifact(*atype*, *name*)
Check if a registered artifact of type *atype* with name *name* exists

Parameters

- **atype** – component, modifier, script, or config
- **name** – registered artifact name, can use prefix to specify project (separated with ":")

Returns True iff an entry for *name* exists

cleanup()
Saves project data if some has changed, and possibly also updates the project files of all loaded projects if they have changed.

7.3 Projects

For `omni-fig` to recognize a directory as a project, it must contain a yaml file named `.fig.yml` (or similarly, see code for all options). This yaml file should contain all project specific information (similar to profiles) that may be useful for running code or interacting with other projects. The most important information contained in the project yaml file:

- `related` - a list of project names that should be loaded before loading this one (all the paths to all related projects must be found in the profile's `projects` table)
- `src` - the relative path to the python file that should be run in order to fully load all components of this project
- `name` - while not enforced, it is strongly encouraged that project info files contain their own name (ideally the same as is used in the profile's `projects` table)
- `project_type` - the registered name of the project type to use when instantiating this project
- `ptype_src_file` - a python file to run before trying to load this project (for example, as it might define and register the desired project type)
- `py_info` - relative path to a python file that defines project meta data (this is particularly useful for projects that are packages, as the meta data is potentially already specified in various places, like `requirements.txt`) (`omni-fig` itself is a good example, see this project's `.fig.yml` file and the associated `omnifig/_info.py` file)

The directory that contains the project info file (`.fig.yml`) is defined as the "project directory". All paths in the info file should be relative to the project directory. By default, if there is a directory called `config` or `configs` in the project directory, then that directory will automatically be registered as a config directory (ie. all yaml files inside will be registered while preserving the folder hierarchy) - see the unit tests (in `test/` for an example).

When a project is loaded, first the desired type is identified. As a result, you can subclass the `Project` class and override the behavior of project objects. Note that this is a fairly advanced featured and should be used only when absolutely necessary (atm I'm not sure why I added this feature in the first place).

class Cerifiable
Bases: `object`

__certify__(*A*, ***kwargs*)

class Workspace(*silent=False*, ***kwargs*)
Bases: `omnifig.containers.Container`

__init__(*silent=False*, ***kwargs*)

_process(*raw*)

initialize()
This loads the project, primarily by registering any specified config files, importing specified packages, and finally running any provided source files

Returns None

load_configs(*paths=[]*)
Registers all specified config files and directories

load_src(*srcs=[]*, *packages=[]*)
Imports all specified packages and runs the specified python files

meta_rules()

meta_rules_fns()

reset_registries()
Clears all registries

register_artifact(*atype*, *name*, *info*)
General function to register an artifact of type *atype* with name *name*

register_script(*name*, *fn*, *description=None*, *use_config=False*)
Function to register a script

Parameters

- **name** – name of script
- **fn** – script function (usually a callable that expects the config object)
- **use_config** – `True` if the config should be passed as only arg when calling the script function, otherise it will automatically pull all arguments in the script function signature
- **description** – a short description of what the script does

Returns

register_component(*name*, *fn*, *description=None*)
fn takes a single input - a Config object The config object is guaranteed to have at least one entry with key "_type" and the value is the same as the registered name of the component.

Parameters

- **name** – str (should be unique)
- **fn** – callable accepting one arg (a Config object) (these should usually be classes)
- **description** – description of what this component is about

register_modifier(*name*, *fn*, *description=None*, *expects_config=False*)
fn takes as input a component and a Config object.

Parameters

- **name** – str (should be unique)
- **fn** – callable accepting one arg (the "create_fn" of a registered component) (these should usually be classes)
- **description** – description of what this modifier is about

register_config(*name*, *path*)
Register a file as a named config

Parameters

- **name** – str (should be unique)
- **path** – full path to the config

register_config_dir(*path*, *recursive=False*, *prefix=None*, *joiner='/'*)
Registers all yaml files found in the given directory (possibly recursively)

When recusively checking all directories inside, the internal folder hierarchy is preserved in the name of the config registered, so for example if the given `path` points to a directory that contains a directory `a` and two files `f1.yaml` and `f2.yaml`:

Contents of `path` and corresponding registered names:

- `f1.yaml` => `f1`
- `f2.yaml` => `f2`
- `a/f3.yaml` => `a/f3`
- `a/b/f4.yaml` => `a/b/f3`

If a `prefix` is provided, it is appended to the beginning of the registered names

Parameters

- **path** – path to root directory to search through
- **recursive** – search recursively through sub-directories for more config yaml files
- **prefix** – prefix for names of configs found herein
- **joiner** – string to merge directories when recursively searching (default `/`)

Returns None

has_artifact(*atype*, *name*)
Check if this workspace has an artifact of type *atype* registered with name *name*

find_artifact(*atype*, *name*)
Find a registered artifact from the type and name

Parameters

- **atype** – component, modifier, script, or config
- **name** – registered name

Returns artifact entry, throws UnknownArtifactError if *atype* is not recognized,

and the corresponding artifact missing error if the name cannot be found

view_artifacts(*atype*)
Return a shallow copy of the full registry of type *atype*

has_script(*name*)

find_script(*name*)

view_scripts()

has_component(*name*)

find_component(*name*)

view_components()

has_modifier(*name*)

find_modifier(*name*)

view_modifiers()

has_config(*name*)

find_config(*name*)

view_configs()

run (*script_name=None*, *config=None*, ***meta_args*)
This actually runs the script given the `config` object.

Before starting the script, all meta rules are executed in order of priority (low to high) as they may change the config or script behavior, then the run mode is created, which is then called to execute the script specified in the config object (or manually overridden using `script_name`)

Parameters

- **script_name** – registered script name to run (overrides what is specified in `config`)
- **config** – config object (usually created with `get_config()` (see *Config System*)
- **meta_args** – Any additional meta arguments to include before running

Returns script output

create_component (*info*)
Creates the component specified in info (checks component registry using info.pull('_type'), and modifier registry for info.pull('_mod'))

_mod can be a list, in which case they will be applied in the given order, eg:

let mods = [A, B, C]

component <- C(B(A(component)))

_mod can also be a dict, in which case the keys should be the mod names and the values the order (low to high). So for the same behavior as above, a _mod could also be {A:0, B:1, C:2}

NOTE: generally, modifiers should be ordered from more specific to more general

Parameters **info** – should be a config object with attribute "_type" (and optionally "_mod")

Returns component created using the provided config (`info`)

process_argv (*argv=()*, *script_name=None*)
Parses the command line arguments to identify the meta arguments, script name (optionally overridden using `script_name`), and config args

From that, this builds the config and meta config object.

Parameters

- **argv** – list of all command line arguments to parse in order
- **script_name** – optional script name to override any script specified in `argv`

Returns config object (containing meta config under `_meta`)

_load_config_from_path (*path*, *process=True*)
Load the yaml file and transform data to a config object

Generally, `get_config` should be used instead of this method

Parameters

- **path** – must be the full path to a yaml file
- **process** – if False, the loaded yaml data is passed without converting to a config object

Returns loaded data from path (usually as a config object)

_merge_configs (*configs*, *parent_defaults=True*)
configs should be ordered from oldest to newest (ie. parents first, children last)

This is an internal method used by `get_config()` and should generally not be called manually.

_process_single_config(*data*, *process=True*, *parents=None*, *tree=None*, *me=''*)
This loads the data (if a path or name is provided) and then checks for parents and loads those as well

Generally, `get_config` should be used instead of this method

Parameters

- **data** – config name or path or raw data (dict/list) or config object
- **process** – configurize loaded data
- **parents** – if None, no parents are loaded, otherwise it must be a dict where the keys are the absolute paths to the config (yaml) file and values are the loaded data

Returns loaded data (as a config object or raw)

create_config(**contents*, ***parameters*)
Top level function for users. This is the best way to load/create a config object.

All parent config (registered names or paths) that should be loaded must precede any manual entries, and will be loaded in reverse order (like python class inheritance).

If the key `_history_key` is specified and not `None`, a flattened list of all parents of this config is pushed to the given key.

Parameters

- **contents** – registered configs or paths or manual entries (like in terminal)
- **parameters** – specify parameters manually as key value pairs

Returns config object

class Project(*profile=None*, ***kwargs*)
Bases: *omnifig.organization.Workspace*

Projects are used to group code into packets with specific config files that should be loaded all together. A project must contain a yaml file named `.fig.yml` in the root directory of the project (aka the "project directory"), and all paths in that yaml file should be relative to the project directory.

Generally there are two kinds of projects: "packages" and "loose" projects. Package projects are meant to be installed and used as a library, while loose projects may just be a series of python files.

This class may also be subclassed to change the behavior of projects (such as changing the loading), in fact, any subclasses of this class can automatically be registered when providing a name for the project type in the class definition.

classmethod __init_subclass__(*name=None*)
Subclasses can automatically be registered if a `ptype` for the registry is provided

__init__(*profile=None*, ***kwargs*)

get_profile()

get_path()
Gets the path to the project directory

get_related()
Returns a list of project names of all projects that should be loaded prior to this one

static check_project_type(*raw*)
Based on raw project info, check if this project expects a certain project type, if so optionally provide a path to the source file of the project type.

Parameters **raw** – raw project info

Returns None or tuple with project type identifier and path to source file (or None)

_process(*raw*)
Given the raw info loaded from a yaml file, this function checks integrates the information into the project object (self).

Parameters **raw** – dictionary of info (usually loaded from a yaml)

initialize()

register_artifact(*atype*, *name*, *info*, *include_global=True*)

has_artifact(*atype*, *name*, *check_global=True*)

find_artifact(*atype*, *name*, *check_global=True*)

has_config(*name*)

find_config(*name*)

view_artifacts(*atype*)

7.4 Meta Rules

Meta rules allows changing any script's behavior before it is run, primarily by making changes in the loaded config or changing the run mode that will be used for execution. Meta rules can be activated from the command line using the registered code (usually a single letter) preceded by a single "-". A meta rule can also be provided with additional required arguments (ie. strings), but the number of arguments must be specified when registering. If a meta rule is activated it should appear in the meta config (under the branch `_meta` in the config object).

Before a script is executed, all registered meta rules are executed in order of priority (low to high). Since all meta rules are always executed, each rule is expected to check the meta config object whether it has been activated and act accordingly. Note that every meta rule is given the loaded config (and separately the meta config) and returns the config after making whatever changes are desired.

Meta_Rule(*name*, *priority=0*, *code=None*, *expects_args=0*, *description=None*)
Decorator used to register meta rules.

Parameters

- **name** (`str`) – Name of the meta rule (and key in meta config for any required arguments)
- **priority** (`int`) – priority of this meta rule (the higher it is the earlier it is run relative to other meta rules)
- **code** (`Optional[str]`) – Command line code (usually a single letter) preceded by "-" to activate this meta rule
- **expects_args** (`int`) – number of command line arguments required when activating
- **description** (`Optional[str]`) – short description of what this rule does for the help message

Returns decorator function to register the meta rule

register_meta_rule(*name*, *fn*, *priority=0*, *code=None*, *expects_args=0*, *description=None*)
Meta Rules are expected to be a callable of the form:

config <= fn(meta, config)

where `meta` is the meta config, and the output should be the processed `config`

Parameters

- **name** – name of the new rule
- **fn** – callable to enforce rule
- **priority** – priority of this rule (rules are enforced by priority high to low)
- **code** – (optional) single letter code used to activate rule from the terminal
- **expects_args** – number of arguments expected by this meta rule through the terminal
- **description** – one line description of what the rule does

Returns None

7.4.1 Help Rule

As a particularly useful example of how meta rules can be used, the "help rule" implements the help messages for the `fig` command, which includes printing out a list of all registered scripts (after the relevant projects are loaded) as well as descriptions (if they are provided).

Note that by subclassing and reregistering this rule, the help message and behavior can easily be augmented.

help_message(*meta*, *config*)

When activated, this rule prints out help message for the `fig` command, which includes a list of all registered scripts, meta rules, and configs that have been loaded.

Parameters

- **meta** – meta config object
- **config** – config object

Returns [system exit, with code 0]

7.5 Run Modes

The run mode is a component that is created using the meta config object. Aside from any user defined functionality, the run mode responsible for identifying the script that should be run (by default saved to `script_name` in the meta config), find the corresponding function (usually using `self.get_script_info()`), execute it with the provided config, and finally return the output.

class Run_Mode(*A*)

Bases: `object`

Run modes are used to actually execute the script specified by the user in the run command.

It is recommended to register all run_modes with a `run_mode/` prefix, but not required.

__init__(*A*)

static get_script_info(*script_name*)

Given the name of the registered script, this returns the corresponding entry in the registry

run(*meta*, *config*)

When called this should actually execute the registered script whose name is specified under meta.script_name.

Parameters

- **meta** – meta config - contains script_name
- **config** – config for script

Returns output of script

7.5.1 Debug Mode

The debug mode serves as a good example for how run modes can be used. During development of new scripts and components it can be invaluable to use a debugger (such as in pycharm) to step through the code and see exactly where bugs might be lurking. Alternatively, when running in the terminal, you can still debug your script using the `ipdb.post_mortem()` debugger.

The debug mode is activated using a meta rule (`-d => debug`), which then changes the run mode to the debug run mode (registered under `run_mode/debug`. Finally, the debugger also automatically updates the config to include a `debug` config (registered as `debug`)

debug_rule(*meta*, *config*)
When activated, this rule changes the run mode to `run_mode/debug` and updates the config to include

Parameters

- **meta** – meta config object
- **config** – full config object

Returns config object updated with debug config (if available)

class Debug_Mode(*A*)
Bases: *omnifig.modes.Run_Mode*

Behaves just like the default run mode, excapt if the script raises an exception, either the `ipdb post_mortem()` debugger is activated, or if this is already running in a debugger (checked using `sys.gettrace() is not None`) then `ipdb` is not activated.

run(*meta*, *config*)
Calls `ipdb` to activate debugger if an exception is raised when running the script.

7.6 Utilities

While this doesn't include all of the utilities used for organization and managing projects and scripts, this should give you a sense for where some of the functionality from behind the scenes actually originates.

Generally, it should not be necessary for a user to call any of these utilities, but they may be useful to add or change the behavior of `omni-fig`.

include_files(**paths*)
Executes all provided paths to python files that have not already been run.

Parameters **paths** – paths to python files to be executed (if not already)

Returns None

include_package(**packages*)
Imports packages based on their names

Parameters **packages** – list of package names to be imported

Returns None

register_project_type(*name*, *cls*)
Project types allow users to customize the behavior of project objects

Parameters

- **name** – identifier of this project type
- **cls** – project type class

Returns None

get_project_type(*name*)
Gets the project type associated with that name, otherwise returns None

view_project_types()
Returns a copy of the full project type registry

set_profile_cls(*cls*)
Set the class used when loading the profile

get_profile_cls()
Get the class used when loading the profile

set_profile(*profile*)
Set the loaded profile object

load_profile(***overrides*)
Load the profile with the yaml file with the path found with the environment variable `FIG_PROFILE`

Parameters **overrides** – Any additional overrides to use instead of checking the environment variables

Returns loaded profile object

get_profile(***overrides*)
Returns current profile (which gets loaded if there is None)

class set_current_project(*project=None*)
Bases: `object`

Context manager to change set the current project in the context

clear_current_project()
Unset the current project (setting it to None)

CHAPTER

EIGHT

ERRORS

Here is a list of all the `omni-fig` specific errors that may be raised.

exception NoValidProjectError(*ident*)
Bases: `Exception`

Raised when no project is found for the given identifier (which should be the name or path to the project)

exception AmbiguousRuleError(*code*, *text*)
Bases: `Exception`

exception UnknownArtifactError
Bases: `Exception`

exception MissingArtifactError(*atype*, *name*)
Bases: `Exception`

exception MissingComponentError(*name*)
Bases: *omnifig.errors.MissingArtifactError*

exception MissingModifierError(*name*)
Bases: *omnifig.errors.MissingArtifactError*

exception MissingConfigError(*name*)
Bases: *omnifig.errors.MissingArtifactError*

exception MissingScriptError(*name*)
Bases: *omnifig.errors.MissingArtifactError*

exception ConfigNotFoundError(*ident*)
Bases: `Exception`

Raised when a config parameter is not found and no viable defaults are provided

exception MissingParameterError(*key*)
Bases: `Exception`

Raised when a config parameter was not found, and no viable defaults were provided

exception InvalidKeyError
Bases: `Exception`

Only raised when a key cannot be converted to an index for a `ConfigList`

exception UnknownActionError
Bases: `Exception`

Raised when trying to record an unrecognized action with the config object

exception PythonizeError(*obj*)
Bases: Exception

Raised when an object is unable to be turned into a yaml object (primitives, dicts, lists)

exception WrongInfoContainerType(*ctype*, *ctype_src=None*)
Bases: Exception

Raised when trying to load a container, but the container expects a different type (ie. subclass).

get_ctype()

get_mtype_src()

exception ConfigurizeFailed
Bases: Exception

Raised when trying to configurize an object by type, but it ends up not working

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX

Symbols

A

C

D

E

F