omnifig

Felix Leeb

May 14, 2024

INTRODUCTION

`omni-fig` is a lightweight package to help you organize your python projects to make everything clear and easy to understand to collaborators and prospective users, while also offering unparalleled features to accelerate development.

The general-purpose project structure is well suited for both small and large projects, and is designed to be easily extensible to fit your needs. Most importantly, with the powerful configuration system, you never have to worry about any boilerplate code to parse command line arguments, read config files again, or even import the desired project components ever again!

CHAPTER

ONE

USER GUIDE

There are two main parts of the user guide are the project organization and project configuration. The project organization chiefly concerns the expected file structure for `omni-fig` to correctly recognize and load your project, and a variety of different ways you can run your scripts. Keep in mind that most of project structure suggested here is not strictly required, and can easily be adapted to your specific workflow. The project configuration covers all the powerful features the configuration system of `omni-fig` offers.

1.1 Installation

Everything is tested with Python 3.7 on Ubuntu 20.04 and Windows 11, but in principle it should work on any system that can handle the dependencies.

The easiest way to install the stable release is with pip:

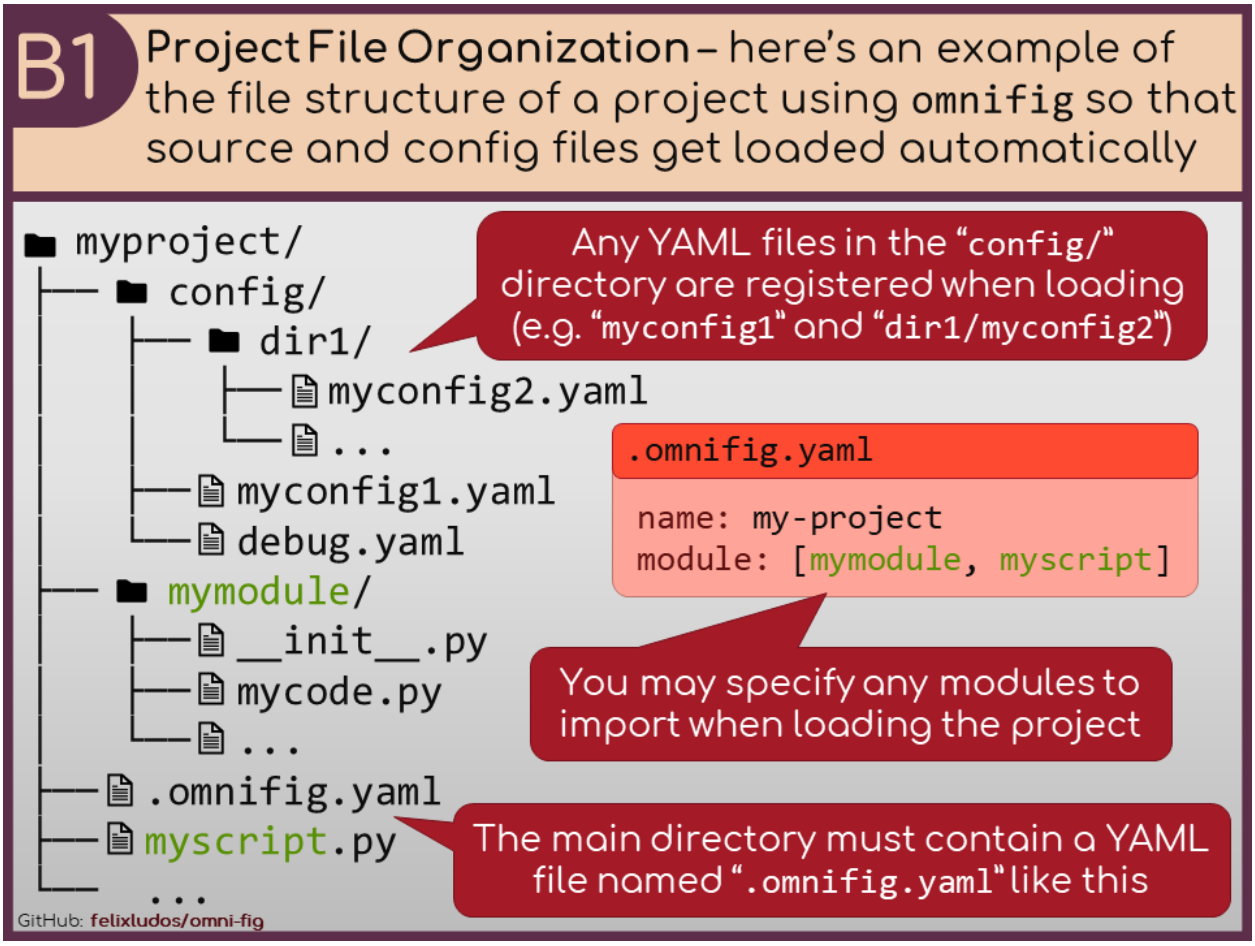
```
pip install omnifig
```

If you want to explore the examples or contribute to the project, you can install the latest version by cloning the repository and installing it with pip: If you want to explore the examples or want to contribute to the project, you can clone the repository and install the package in development mode:

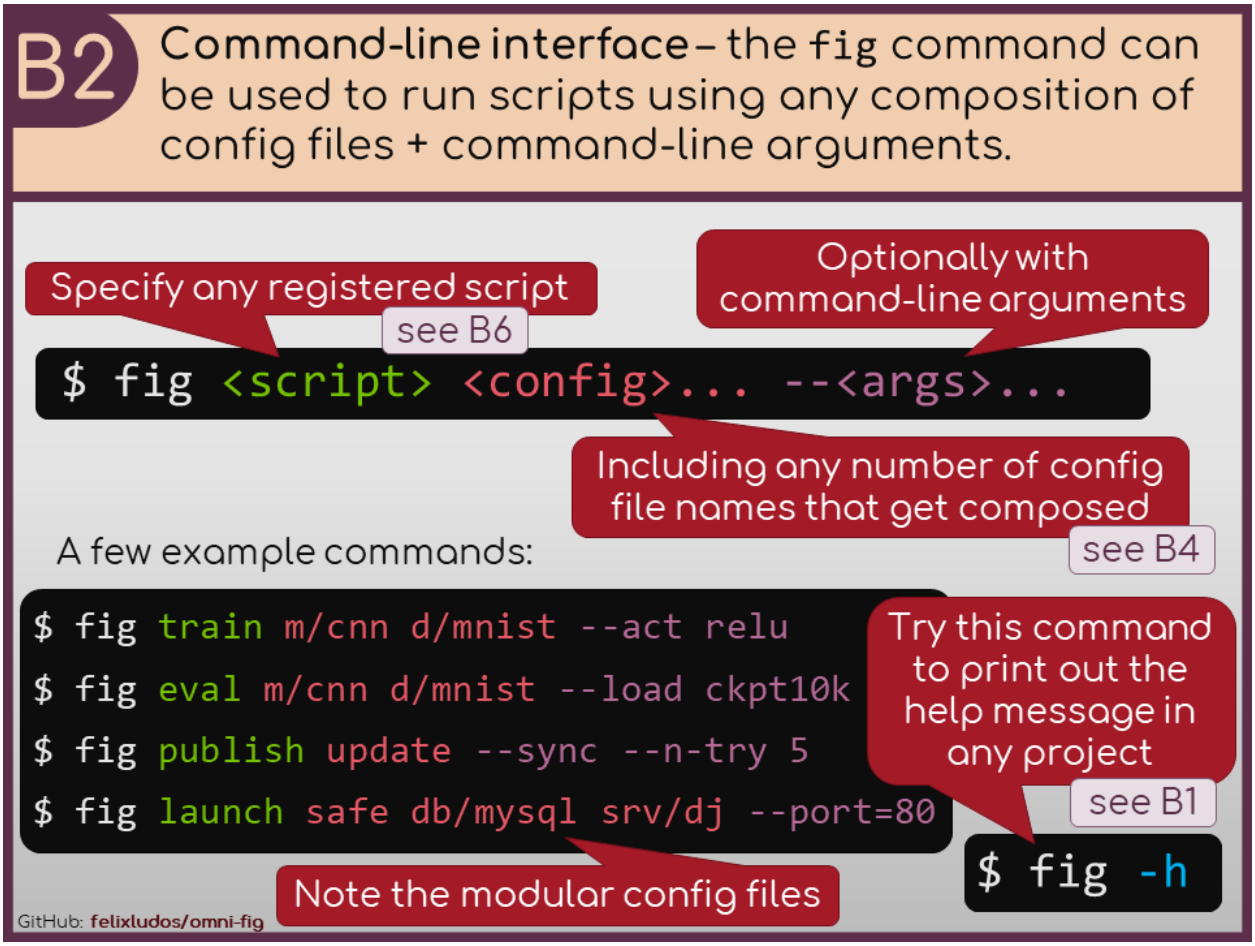
```
git clone https://github.com/felixludos/omni-fig
pip install -e ./omni-fig
```

1.2 Key Features

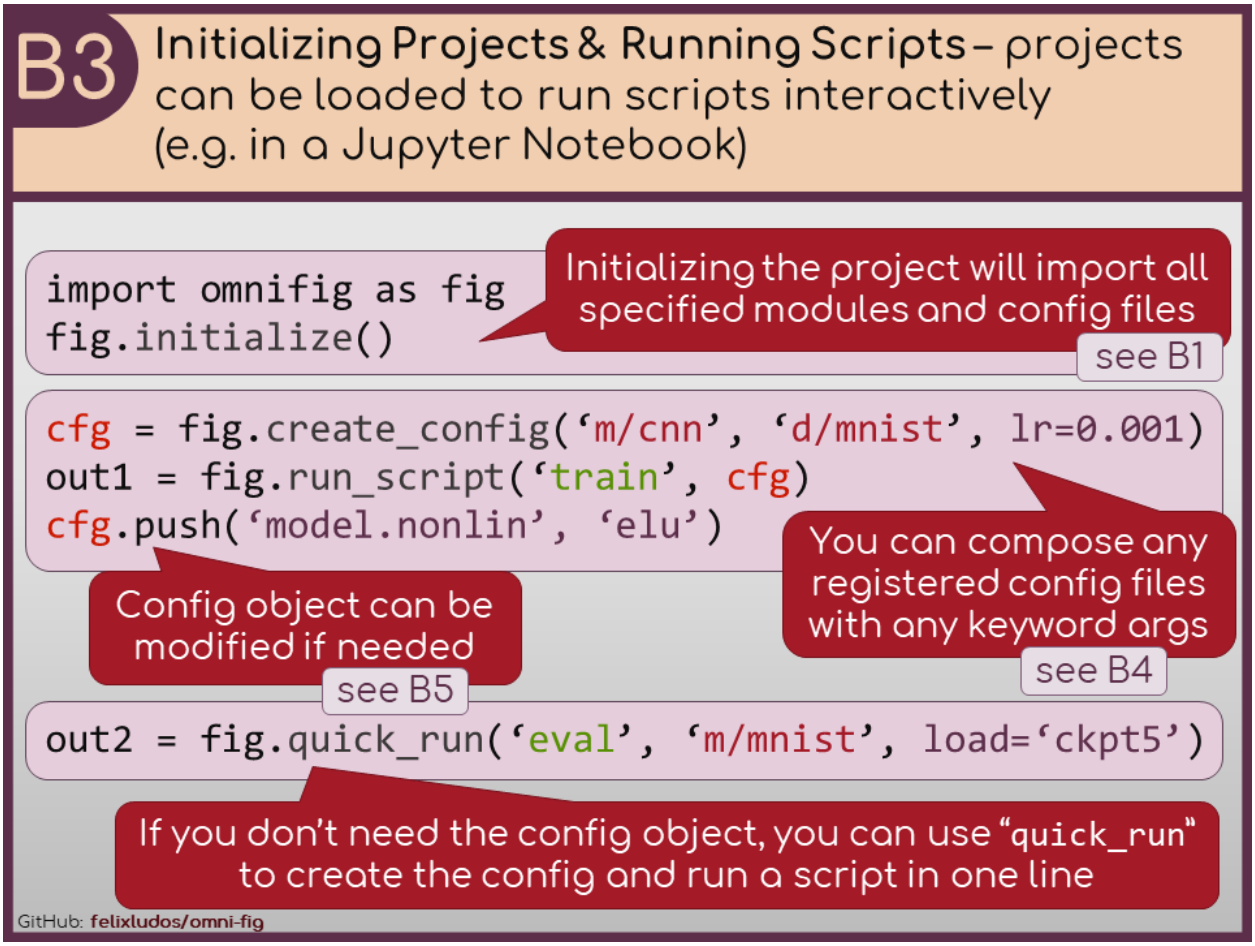
Here is an overview of some of the core features of `omni-fig` with links to more detailed guides.



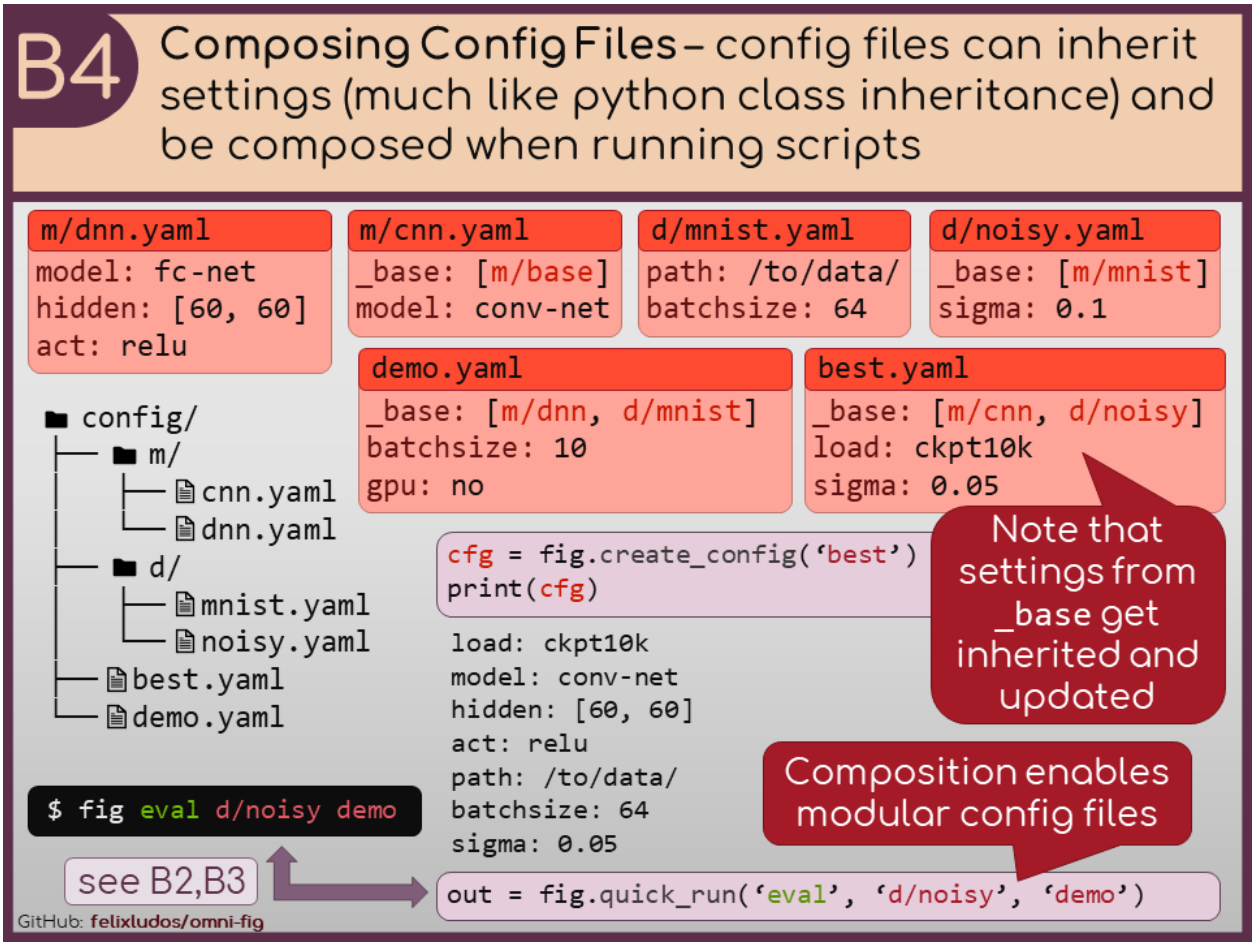
Check out the *file structure* of projects.



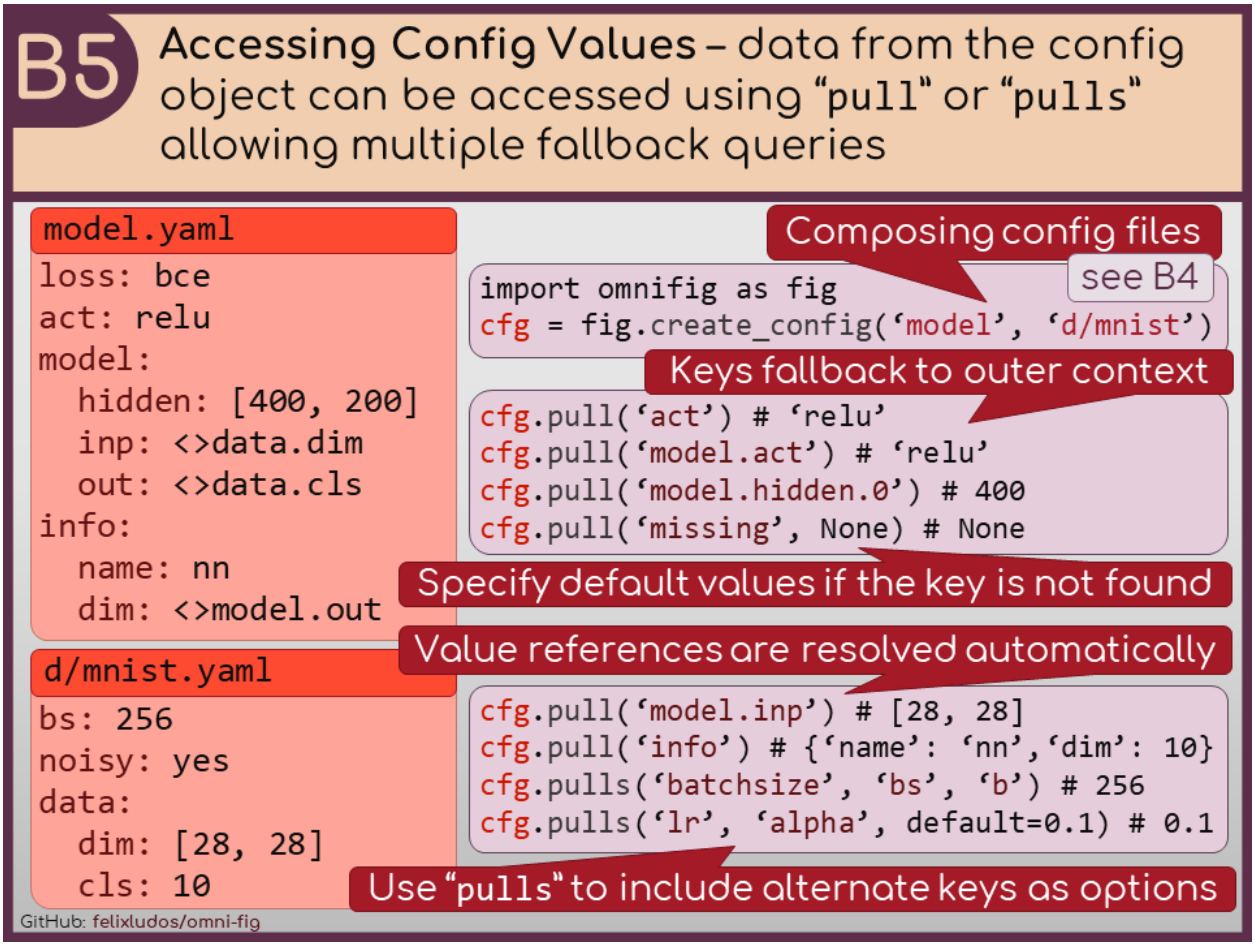
Read more about the *command line interface*.



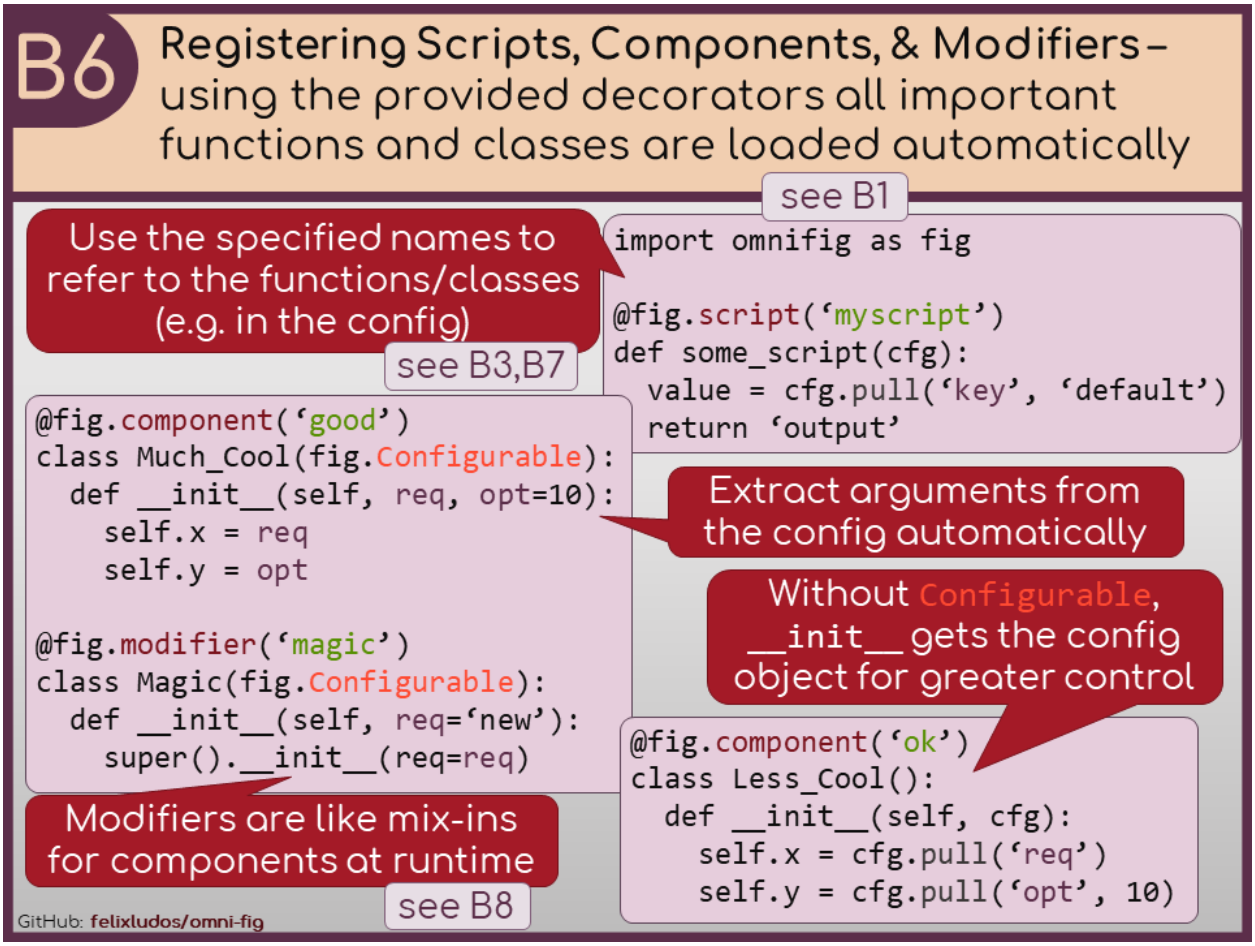
Learn more about using `omni-fig` in *interactive programming environments*.



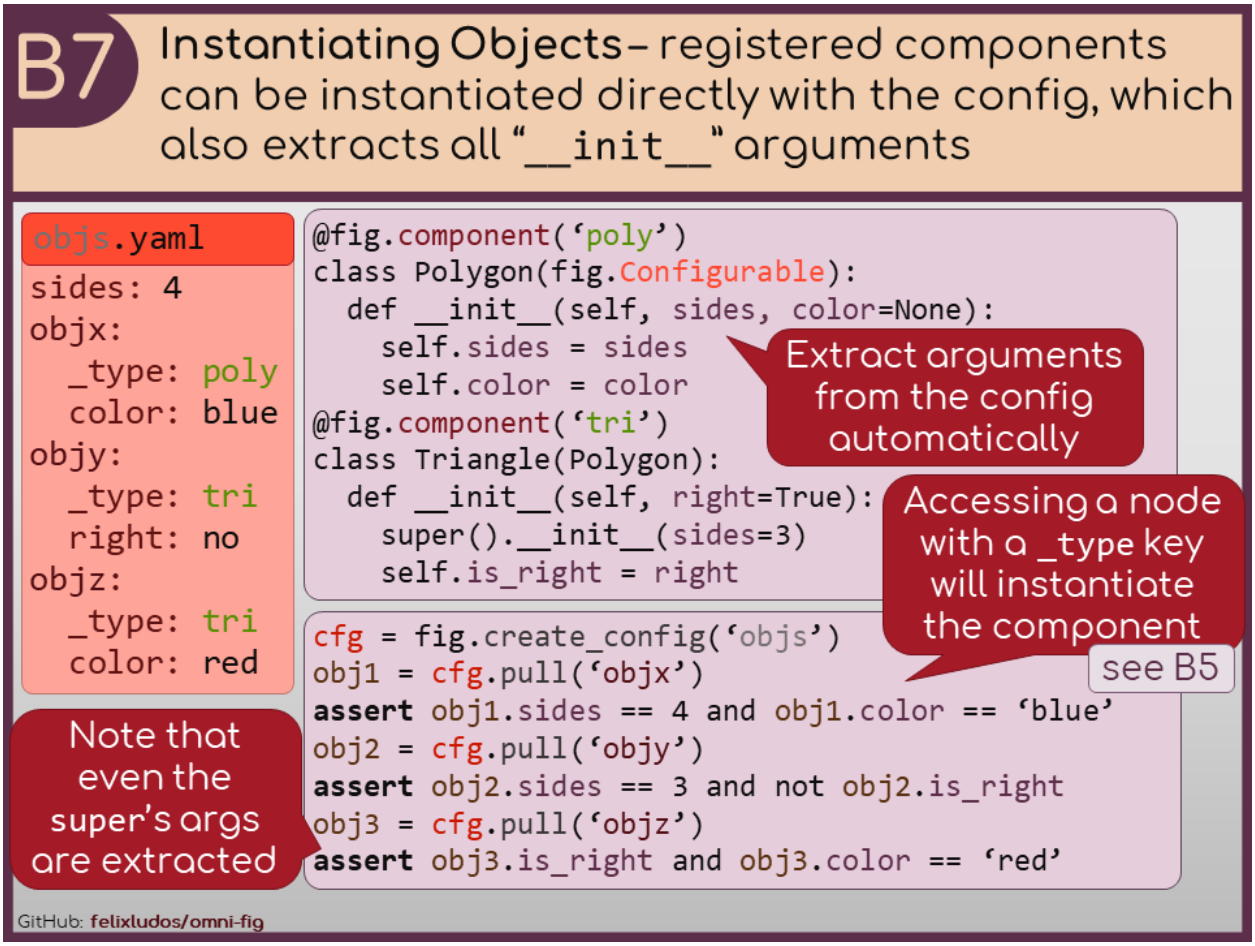
Learn more about *composing configuration files*.



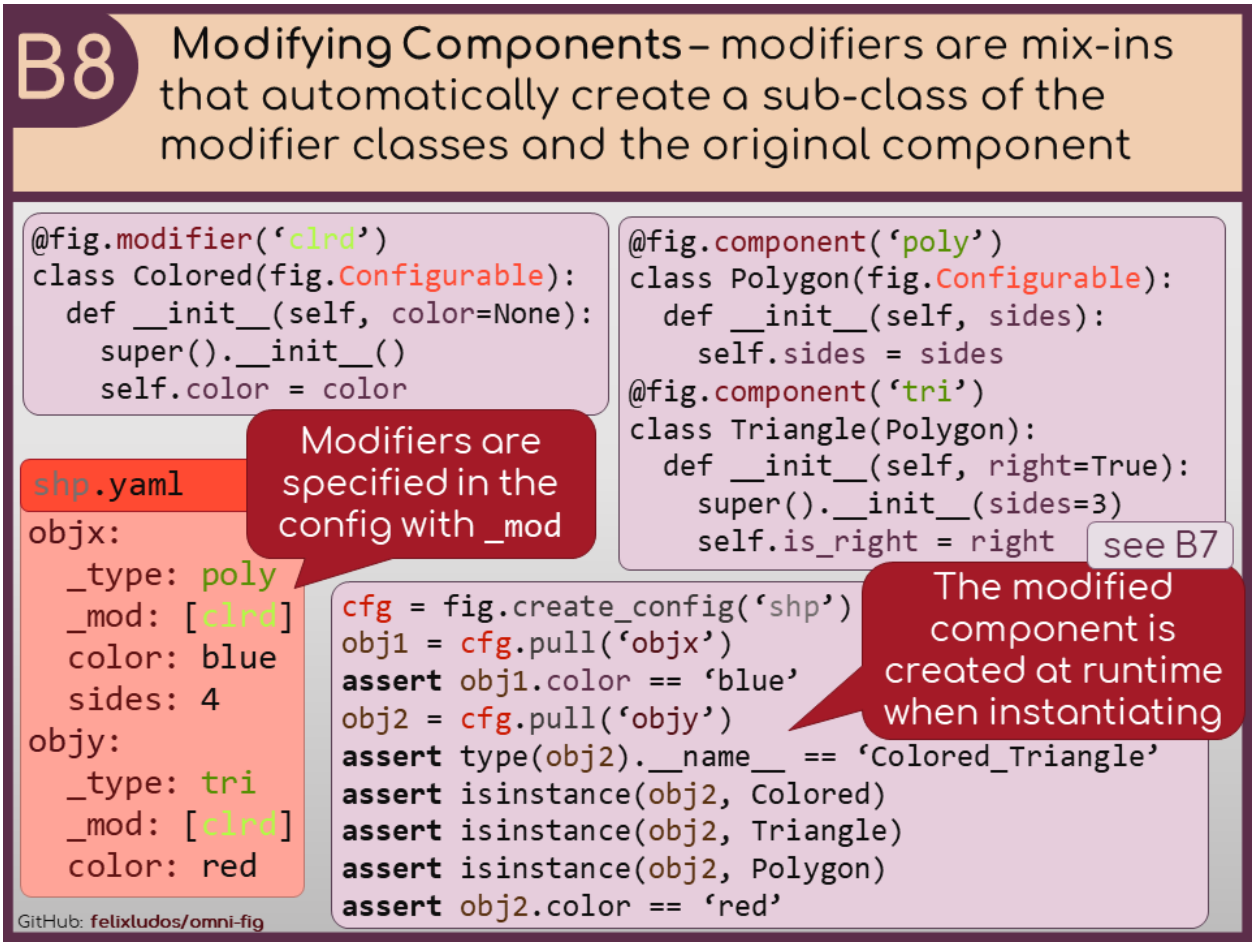
Read more about how to *access config values*.



Check out the guide for details on *project components*.



See the power of automatically *instantiating objects* from the config.



Learn more about *modifying components*.

1.3 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 Github Actions). 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 *profiles*, *projects*, and *behaviors*), 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.

1.3.1 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. A registered `script` (any callable) can be run in a variety of ways (e.g. from the command-line or a in an interactive environment like jupyter) but essentially act as a self contained operation. A `component` is some piece of code (usually a class) that builds an arbitrarily complex object which can be instantiated automatically from the config. Finally, `modifiers` allow `components` to be modified dynamically (see the *guide*). 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. Consequently, as long as all source files are loaded when the project is loaded, all functionality that the developer explicitly wants to make accessible, is accessible by the name at runtime.

Another underrated 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). Furthermore, object serialization (both for persistence and multi-processing) is much easier when the objects are referred to by their registered names.

Check out the *user guide*.

1.3.2 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.

Check out the *user guide*.

1.4 Overview

The project organization chiefly concerns the expected file structure for `omni-fig` to correctly recognize and load your project, and a variety of different ways you can run your scripts (e.g. command-line interface, in a jupyter notebook). Keep in mind that most of project structure suggested here is not strictly required, and can easily be adapted to your specific workflow.

Each project has several registries to keep track of all associated config files, scripts, and classes, as well as the project's root directory. The registries keep track of all top-level functionality you want to make accessible to the user.

1.4.1 File Structure

To create a project, the only thing that is actually required is to have a project info file named `.omnifig.yml` in the root directoy (which may be empty). That project info file may contain the following fields (in YAML format):

- `name` - The name of the project. If not specified, the name will be the name of the directory containing the project info file.
- `module` - Specifies one or multiple python modules that should be imported when the project is loaded. These should be specified as if you were using an import statement, e.g. `mymodule.myscript` instead of `mymodule/myscript.py`. If there are multiple modules to import, they should be specified as a list of strings.
- `src` - (advanced feature) Specifies one or multiple paths to python source files (relative to the project root directory) that should be run when the project is loaded. If there are multiple paths to import, they should be specified as a list of strings. Note that the difference between `module` and `src` is that `module` uses import which adds the module/s to `sys.modules`, while `src` does not. If you are unsure which to use, generally if you would use `import` in your code, use `module`, while if you want to specify the file or directory by its path, use `src`.
- `related` - (advanced feature) Specifies a list of names of other projects to load when this project is loaded. This is useful if you have a project that is a collection of other projects, and you want to load them all at once. Importantly, the projects must be specified by the same name that is used in the *profile*.

By default, if there is a directory called `config` in the project directory, then all yaml files inside will be registered while preserving the directory structure.

So for example, a project directory may look like this:

```
myproject/ - project root directory
|
├── config/ - any YAML config files that should automatically be registered
|   ├── dir1/
|   |   ├── myconfig2.yaml - will be registered as 'dir1/myconfig2'
|   |   └── ...
|   |
|   ├── myconfig1.yaml - will be registered as 'myconfig1'
|   └── debug.yaml - config to be used in :ref:`Debug` mode
|
├── scripts/ - some python module containing all the code
|   ├── __init__.py - required by python to make this a module
|   ├── myscript.py
|   └── ...
|
├── .omnifig.yml - project info file
└── ...
```

Where the `.omnifig.yml` file may look like this:

```yaml
name: some-name
module: scripts
```

See the feature slide *B1*.

To see exactly what else projects can do, check out the documentation on *projects* and their *default behavior*.

1.5 Command Line Interface

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.

The *fig* command should be used like this:

```
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 Behavior that should be activated to modify the execution of the script (and must use the prefix `-`. For example, use `-h` to see the *Help* message, or `-d` to run the script in *Debug* mode.
- `configs` - is an ordered list of names of registered config files that will be composed and passed to the script function.
- `args` - any manually provided arguments to be added to the config object. 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`.

Even if the script name is specified in the config (under the key `_meta.script_name`), you must include an underscore `_` in the command-line command.

However, the `fig` command really just calls *`fig.entry()`*, so you can customize the entry point as well. For example, with a python file `main.py` in the project directory that looks like this:

```
import omnifig as fig

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

Now, running something like `python main.py <script> [<configs>...]` is equivalent to `fig <script> [<configs>...]`. This is useful if you want to add additional functionality to the entry point of your project, or if you want to specify the script to run in the python file instead of the command line. For example, given that we registered a script called `launch-server`, we could create another python file `launch.py` that looks like this:

```
import omnifig as fig

if __name__ == '__main__':
    fig.entry('launch-server')
```

Now running something like `python launch.py [<configs>...]` is equivalent to `fig launch-server [<configs>...]`.

See the feature slide *B2*.

1.5.1 Execution Sequence

Here's a breakdown of exactly what happens when you run the `fig` command, which is the main entry point for running scripts.

1. First, the `omnifig` package is imported.
2. Then *`fig.entry()`* is called.
    1. The profile is detected and loaded (see *Profiles*) with `profile.activate()`.
        1. The current project is detected (see Projects), but not loaded yet.
        2. Any specified active base projects are loaded (see *Profiles*).
    2. *`fig.main()`* is called with the script name (if one is provided to `entry()` and the system arguments `sys.argv`.
        1. The project is loaded, importing any specified modules and running any source files (see Projects) with `project.activate()`.
        2. All registered behaviors are instantiated (see *Behaviors*).
        3. The provided arguments are parsed with the project's `ConfigManager.parse_argv` and the behaviors to produce the config object
        4. Using the config object, the project is validated using `project.validate(config)` method (which allows the config or behaviors to switch projects before the script is run).
        5. The script is run with the config object using `project.run_script(script, config)`.
            1. If a script was provided manually, that is added to the config object.
            2. *`pre_run()`* method is called on all behaviors.
            3. The script is run with the config object
            4. *`post_run()`* method is called on all behaviors.
        6. The project is cleaned up using `project.cleanup()` method.
        7. The output of the script is returned by *`fig.main()`*, but not by *`fig.entry()`*.

1.6 Jupyter Environments

You can also use `omni-fig` in an interactive programming environment such as Jupyter or IPython. This is useful for quickly prototyping and debugging scripts. First, to load a project, use the *`fig.initialize()`* function. Once the project is initialized you can create a config with any of the registered configs using *`fig.create_config()`*. Additionally, you can run any registered scripts with *`fig.run_script()`*, or *`fig.run()`* if the script is already specified in the config.

Remember that when running a *script*, the first positional argument is the config, however you can manually include additional positional keyword arguments which are passed to the script as well. Alternatively, for additional convenience, you can use the *`fig.quick_run()`* function to create a config object and run a script in one line.

See the feature slide *B3*.

1.7 Registration

Aside from config files, `omni-fig` primarily keeps track of three different kinds of top-level deliverables: scripts, components, and modifiers. These are all registered using the corresponding decorators *fig.script()*, *fig.component()*, and *fig.modifier()*.

- Scripts are functions that should expect the first positional argument to be the config object.
- Components are classes that are recommended to subclass *fig.Configurable* (see *Configurable*) to extract all the arguments in `__init__` from the config object automatically. If they do not subclass *fig.Configurable*, then they should expect the first positional argument to be the config object.
- Modifiers are classes much like components, except that they are used to modify components by dynamically creating a subclass of the modifier and the component at runtime. Consequently, it is also strongly recommended that modifiers subclass *fig.Configurable*, or otherwise they should expect the first positional argument to be the config object.

For simple components and scripts (especially components which are functions), there are two convenience types called *fig.autocomponent* and *fig.autoscript* respectively. These variants behave the same as the regular decorators, except that instead of passing the config object during initialization, the arguments of the registered class or function are extracted from the config object automatically (much like *fig.Configurable*).

See the feature slide *B6*.

1.7.1 Configurable

When registering classes as components and modifiers, it is strongly recommended that the class is a subclass of *fig.Configurable*. This streamlines the *object instantiation* from the config, so that all arguments in the `__init__`, are automatically extracted from the config.

For additional control on how arguments are extracted from the config, checkout the *fig.config_aliases* and `fig.config_silence` decorators.

```python
import omnifig as fig

class Shape(fig.Configurable): # note that does not get registered
    def __init__(self, area, color):
        self.color = color
        self.area = area

@fig.component('circle')
class Circle(Shape):
    def __init__(self, radius, **kwargs):
        super().__init__(area=3.14*radius**2, **kwargs)
        self.radius = radius


@fig.component('rectangle')
class Rectangle(Shape):
    @fig.config_aliases(width='w', height='h')
    def __init__(self, width, height, **kwargs):
        super().__init__(area=width*height, **kwargs)
        self.width = width
        self.height = height
```

(continues on next page)

(continued from previous page)

```
@fig.component('square')
class Square(Rectangle):
    @fig.config_aliases(side=['size', 's'])
    def __init__(self, side, **kwargs):
        super().__init__(width=side, height=side, **kwargs)
        self.side = side
```

With these components, you can now instantiate them with the config for example:

```
cfg = fig.create_config(_type='circle', color='red', radius=5)
obj1 = cfg.create()

assert obj1.color == 'red'
assert obj1.radius == 5
assert isinstance(obj1, Circle)

obj2 = cfg.create(color='green')

assert obj2.color == 'green'
assert obj2.radius == 5

obj3 = cfg.create(2)

assert obj3.color == 'red'
assert obj3.radius == 2

cfg = fig.create_config(_type='square', color='blue')

obj4 = cfg.create(side=5)

assert obj4.color == 'blue'
assert obj4.side == 5
assert isinstance(obj4, Square)
assert isinstance(obj4, Rectangle)

obj5 = cfg.create(size=6)

assert obj5.color == 'blue'
assert obj5.side == 6

obj6 = cfg.create(s=7, color='yellow')
assert obj6.color == 'yellow'
assert obj6.area == 49
```

Note, that just because a class is a subclass of *fig.Configurable* does not mean you can't continue to instantiate the class as usual. So to continue the example above:

```
obj7 = Circle(3, color='purple')

assert obj7.color == 'purple'
assert obj7.radius == 3
```

(continues on next page)

(continued from previous page)

```
obj8 = Square(4, color='orange')

assert obj8.color == 'orange'
assert obj8.side == 4

obj9 = Rectangle(5, 6, color='black')

assert obj9.color == 'black'
assert obj9.width == 5
assert obj9.height == 6
```

1.7.2 Modifying Components

Modifiers are effectively subclasses and mix-ins for which are abstracted from their super classes (registered components). Unlike regular mix-ins, you don't have to define the classes with all the desired mix-ins beforehand, and instead you can create them dynamically at runtime using the config.

To continue the example above, here are two examples of potential modifiers:

```
@fig.modifier('named')
class Named(fig.Configurable):
    def __init__(self, name=None):
        self.name = name

@fig.modifier('drawable')
class Drawable(Shape):
    def draw(self):
        ...

@fig.modifier('dark')
class Dark(Shape):
    @fig.config_aliases(color='c')
    def __init__(self, area, color):
        color = f'dark-{color}'
        super().__init__(area, color)
```

Now instead of needing to define every combination of `Named`, `Drawable`, and `Shape` beforehand, you can create only the combinations you need dynamically at runtime using the config:

```
cfg = fig.create_config(_type='circle', _mod='named', color='red', radius=5)
obj1 = cfg.create('my-circle')

assert obj1.name == 'my-circle'
assert obj1.color == 'red'
assert obj1.radius == 5
assert isinstance(obj1, Circle)
assert isinstance(obj1, Named)
assert type(obj1).__name__ == 'Named_Circle'

cfg = fig.create_config(_type='square', _mod=['named', 'drawable'], color='blue')
obj2 = cfg.create()
```

(continues on next page)

(continued from previous page)

```
assert obj2.name is None
assert obj2.color == 'blue'
assert type(obj2).__name__ == 'Named_Drawable_Square'

cfg = fig.create_config(_type='square', _mod=['named', 'dark'], c='green', name='my-
↪square')
obj3 = cfg.create()

assert obj3.name == 'my-square'
assert obj3.color == 'dark-green'
assert isinstance(obj3, Dark)
assert type(obj3).__name__ == 'Named_Dark_Square'
```

See the feature slide *B8*.

1.8 Profiles

Generally, every OS image or file system should use it's own *profile*. The profile is specified by defining an environment variable `OMNIFIG_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 to load them remotely. For example, a project can specify related projects as dependencies, which get loaded automatically when the project is loaded, but only if the paths to those dependency projects are listed in the profile info file. 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):

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

1.9 Behaviors

One of the key features of `omni-fig` to provide additional control and customization over how your scripts are run is the ability to specify *behaviors*. Behaviors are classes with a variety of callbacks at different stages of the *execution sequence*. Behaviors are usually activated by including a flag in the command line arguments, but can also be activated directly through in the config under the `_meta` branch.

A comprehensive documentation of the callbacks are found in *`AbstractBehavior`*, but most common callbacks are:

- *`parse_argv`*: called before the `sys.argv` is parsed (and commonly used to check whether or not the behavior should be activated (see *`parse_argv`*).
- *`validate_project`*: called after the project is loaded, and can be used to switch to a different project.
- *`include`*: called after the project is validated, to select which behaviors should be run.
- *`pre_run`*: called before the script is run
- *`handle_exception`*: called when an exception is raised during the script execution

- *post_run*: called after the script is run, including the output of the script. If this callback returns a value, it will be used as the output of the script.

Additionally, `omni-fig` comes with a few simple, common behaviors that serve as examples of what you can do with behaviors. For more details check out the *documentation*.

1.9.1 Help

The *Help* behavior provides a simple way to add help messages to your scripts. It is activated by the `-h` flag, and will print out the help message for the `fig` command. If a script is specified in a command with the help flag, then the `__doc__` of the script function is also included in the help message.

1.9.2 Debug

The *Debug* behavior provides a simple way to add debug messages to your scripts. It is activated by the `-d` flag, and will automatically compose the config object with a registered config named `debug`. This is particularly useful when using IDEs like PyCharm or VS Code where you can define a fixed script execution `fig -d`, and then you can directly edit the file `config/debug.yaml` to specify the script and other config files that should be included.

Note that you can specify which script should be run in the config with the key `_meta.script_name`.

1.9.3 Quiet

The *Quiet* behavior provides a simple way to suppress messages sent to the console. It is activated by the `-q` flag.

1.10 Overview

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 tree-like structure in the config much like a tree where each branch corresponds to the a `dict` or `list` of arguments.

This section discusses different ways to create and use the config object. You can create config objects using the *fig.create_config* by directly passing keyword arguments. However, usually, you will want to populate the config object with parameters in config files, which you can do either by passing in the path to a yaml file to *fig.create_config* as a positional argument, or the recommended way: by the name of the registered config file. All yaml files in the `config/` directory are automatically registered as long as you create a *project info file*. Note that you can pass in any number of positional arguments to *compose multiple config files*. From the *command-line interface*, you can also specify multiple config files and keyword arguments.

The easiest way to access parameters in the config object is using *pull()*, but there's more information on *accessing values here*.

```
import omnifig as fig

cfg = fig.create_config(arg1=True, arg2='hello', arg3=[1,2,3])

assert cfg.pull('arg1') is True
assert cfg.pull('arg2') == 'hello'
assert cfg.pull('arg3') == [1,2,3]
```

(continues on next page)

(continued from previous page)

```
# specify a default when the query is not found
assert cfg.pull('arg4', 'default') == 'default'

# specify multiple queries to check in order with `pulls`
assert cfg.pulls('arg5', 'arg2') == 'hello'
assert cfg.pulls('arg6', 'arg7', 'arg8', default='not-found') == 'not-found'
```

1.11 Composing Config Files

Perhaps the most important feature of the configuration system is that config files can easily be composed to encourage config files to be more modular. Specifically, when loading a config file, it will inherit all values of any config files listed under the key `_base`. 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).

For example, consider the following config directory structure:

```
config/
    base.yaml
    cluster.yaml
    demo.yaml
    model/
        base.yaml
        simple.yaml
        large.yaml
    data/
        base.yaml
        mnist.yaml
        cifar.yaml
```

Where `base.yaml` contains the following:

```
checkpoint-epochs: 5
gpu: no
```

`cluster.yaml` contains the following:

```
_base: [base]
gpu: yes
num-workers: 8
```

`model/base.yaml` contains the following:

```
_base: [base]
optim: sgd
lr: 0.001
act: relu
```

`model/simple.yaml` contains the following:

```yaml
_base: [model/base]
model-name: deep-nn
hidden: [40, 40]
```

`model/large.yaml` contains the following:

```yaml
_base: [model/base]
model-name: large-nn
hidden: [300, 300, 300]
batch-norm: yes
optim: adam
```

`data/base.yaml` contains the following:

```yaml
_base: [base]
batch-size: 128
data-dir: /path/to/all/data
```

`data/mnist.yaml` contains the following:

```yaml
_base: [data/base]
dataset: mnist
num-classes: 10
```

`data/cifar.yaml` contains the following:

```yaml
_base: [data/base]
dataset: cifar
num-classes: 100
```

`demo.yaml` contains the following:

```yaml
_base: [data/mnist, model/simple]
```

Then, the following configs would be the result of composing the above config files:

```
>>> import omnifig as fig
>>> print(fig.create_config('cluster', 'model/simple', 'data/mnist'))
gpu: yes
num-workers: 8
checkpoint-epochs: 5
optim: sgd
lr: 0.001
act: relu
model-name: some-model
hidden: [40, 40]
batch-size: 128
dataset: mnist
data-dir: /path/to/all/data
num-classes: 10

>>> print(fig.create_config('model/large', 'data/cifar'))
gpu: no
checkpoint-epochs: 5
```

(continues on next page)

(continued from previous page)

```
optim: adam
lr: 0.001
act: relu
model-name: large-nn
hidden: [300, 300, 300]
batch-norm: yes
batch-size: 128
data-dir: /path/to/all/data
dataset: cifar
num-classes: 100

>>> print(fig.create_config('model/large', 'data/cifar', 'cluster'))
gpu: yes
num-workers: 8
checkpoint-epochs: 5
optim: adam
lr: 0.001
act: relu
model-name: large-nn
hidden: [300, 300, 300]
batch-norm: yes
batch-size: 128
data-dir: /path/to/all/data
dataset: cifar
num-classes: 100

>>> print(fig.create_config('demo'))
gpu: no
checkpoint-epochs: 5
optim: sgd
lr: 0.001
act: relu
model-name: some-model
hidden: [40, 40]
batch-size: 128
dataset: mnist
data-dir: /path/to/all/data
num-classes: 10
```

See the feature slide *B4*.

1.12 Accessing Config Values

Once the config object is *created*, the primary way to access values in the config is via the *pull()* (while you can update individual values using *push()*). Optionally, you can provide a default value to be returned if the query is not found. If no default value is provided, a *SearchFailed* error is raised (which is subclass of `KeyError`). Additionally, the *pulls()* method allows you to provide any number of fallback queries.

Note: _ and - are interchangeable in config keys

When reading (aka *pulling*) arguments from the config, if an argument is not found in the current branch and the query is not hidden (i.e. it does not begin with _), 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 how deep the node containing the argument actually is. Although this behavior can be changed using the `ask_parents` key in the *config settings*.

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

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

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

When this yaml file is loaded (e.g. `config = fig.create_config('myconfig')`), we could use it like so:

```python
assert config.pull('favorites.language') == 'Python'
assert config.pull('favorites.0') == 'Innvoation'
assert config.pull('app.color') == 'red'
assert config.pull('favorites.activity') == 'Innovation'
assert config.pull('trip.0.location') == 'London'
assert config.pull('trip.1.nights', 4) == 2
assert config.pull('app.publisher', 'unknown') == 'unknown' # default

assert config.pulls('jacket.color', 'wallpaper.color') == 'red'
assert config.pulls('jacket.price', 'price', 'total_cost', default='too much') == 'too␣
↪much'
```

While this example should give you a sense for what kind of features the config system offers, a more comprehensive list of how queries in the config are resolved and the values are processed.

See the feature slide *B5*.

1.12.1 Queries

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

- `'_{}'` - hidden query - is not visible to child branches when they defer to parents
- `push()`/`pull()` `'{1}.{2}'` - *deep* query - equivalent to `['{1}']['{2}']`
- `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

1.12.2 Values

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

- `'<>{}'` - local alias - defer to value of `{}` starting search for the key here
- `'<o>{}'` - (advanced feature) origin alias - defer to value of `{}` starting search 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 - behaves as though this key didn't exist (and doesn't defer to parent)

Currently there are no escape sequences, so any values starting with <> or <o> will be treated as aliases and values that are `_x_` or `__x__` will not be processed as regular strings. However, if necessary, you can easily implement a component to escape these values using the automatic *object instantiation*, like so:

```
@fig.autocomponent('escaped-str')
def escape_str(value):
    return value

cfg = fig.create_config(special={'_type': 'escaped-str', 'value':'<>some-value'})

assert cfg.pull('special') == '<>some-value'
```

1.13 Automatic Instantiation

Beyond *accessing* primitives or simple containers like lists and dicts, you can also create arbitrarily complex *components* automatically with the config object.

All you need is to include the key `_type` which specifies the name of the component that should be instantiated. The config object will then automatically instantiate the component and pass the config node to the component's constructor (unless the component is *configurable*).

See the feature slide *B7*.

Here's an example:

```
import omnifig as fig

@fig.component('cmp1')
class Component1():
    def __init__(self, config):
```

(continues on next page)

(continued from previous page)

```
        self.x = config.pull('x') # will raise an error if 'x' is not found
        self.y = config.pull('y', 10)

@fig.component('cmp2')
class Component2(fig.Configurable):
    def __init__(self, x, y=20):
        self.x = x
        self.y = y

cfg = fig.create_config(
          b1={'_type': 'cmp1', 'x': 'value'},
          b2={'_type': 'cmp2', 'x': 'value2', 'y': 1},
)

print(cfg) # prints out:
# obj1:
#   _type: cmp1
#   x: value
# obj2:
#   _type: cmp2
#   x: value2
#   y: 1

obj1 = cfg.pull('b1') # instantiates a Component1 object

assert obj1.x == 'value'
assert obj1.y == 10
assert isinstance(obj1, Component1)

obj2 = cfg.pull('b2') # instantiates a Component2 object

assert obj2.x == 'value2'
assert obj2.y == 1
assert isinstance(obj2, Component2)
```

Additionally, you can specify *modifiers* with the _mod key, and a custom *creator* with the``_creator`` key for more control over what gets instantiated and how.

1.13.1 Creating/Processing Values

By default, only one instance of each component is created, so when pulling repeatedly, you will get the same instance of the component. However, this behavior can be adjusted by changing the *settings* or by creating instances explicitly using `create`. In contrast, you can explicitly access the value of a node including passing in positional and keyword arguments using `process`. For convenience, there are a few methods that combine the two: `peek_create`, `peek_process` (see for more info about *peeking*). Lastly, you can clear all instantiated components for the whole config tree using `clear_product`.

Continuing the example above:

```
obj3 = cfg.pull('b1') # returns the same object as obj1

assert obj1 is obj3
```

(continues on next page)

(continued from previous page)

```
assert obj2 is cfg.peek_process('b2') # returns the same object as obj2

obj4 = cfg.peek_create('b2') # returns a new object

assert obj2 is not obj4

cfg.clear_product() # clears all instantiated components

assert obj1 is not cfg.pull('b1') # returns a new object

cfg2 = fig.create_config(_type='cmp2', x='value3', y=0)

obj5 = cfg2.process(y=100)

assert obj5.x == 'value3'
assert obj5.y == 100

obj6 = cfg2.process(y=200)

assert obj6.x == 'value3'
assert obj6.y == 100 # not 200 because the component was already instantiated
assert obj5 is obj6

obj7 = cfg2.create('value4', y=200)

assert obj7.x == 'value4'
assert obj7.y == 200 # because the component was created explicitly
assert obj7 is not obj6
```

1.14 Traversing Configs

The config object is always a node in a tree. The root node is what is returned when creating the config node, but when *instantiating* a component the corresponding child node is passed in. You can also traverse the config tree using by iteration or using *peek()*. *peek()* is given a query (usually a key) and then traverses the config object to find the node corresponding to that query.

```
cfg = fig.create_config(x='hello',
    l = [1, 2, 3]
    d = dict(a=1, b=2, c=dict(x=1, y=2))
)

print(cfg) # prints out:
# x: hello
# l: [1, 2, 3]
# d:
#   a: 1
#   b: 2
#   c:
#     x: 1
```

(continues on next page)

(continued from previous page)

```
#     y: 2

assert cfg.parent is None

node = cfg.peek('x')
assert node.parent is cfg
assert node.pull() == 'hello'

node2 = cfg.peek('l')
assert node2.parent is cfg
assert node2.pull() == [1, 2, 3]
assert cfg.peek('x.l') is node2

assert node2.peek('0') is cfg.peek('l.0')
assert node2.peek('1').pull() == 2

d = cfg.peek('d')
assert d.pull('a') == 1
assert d.pull('x') == 'hello'
assert d.pull('c.x') == 1

c = d.peek('c')
assert c.pull('x') == 1
assert c.pull() == dict(x=1, y=2)
assert c.parent is d

assert c.root is cfg
assert cfg.root is cfg

assert not cfg.is_leaf
assert not d.is_leaf
assert c.is_leaf
```

Similar to *pull()* (used to *access* config values), *peek()* also has a variant which allows for multiple queries *peeks()*.

1.14.1 Iteration

You can iterate over the child nodes of a config nodes either using *peek_children* or *pull_children* (to iterate over the values).

There are variants to include the keys when iterating *pull_named_children* and *peek_named_children*.

1.15 Config Settings

There are a few settings you can choose to affect the behavior of the config object. You can view and change the settings with the property `cfg.settings`:

- `silent` - If `True`, suppresses all print messages when values from the config are accessed and updated (`False` by default).
- `readonly` - If `True`, prevents any changes to the config values (`False` by default).
- `creator` - If specified, this must be the name of a registered creator, which will be used to create the config values (defaults to *`DefaultCreator`*), for more information see the guide about *creators*.
- `force_create` - If `True`, will always *create* new instances of components when pulling config values (`False` by default).
- `allow_create` - If `False`, only instances of components that have already been instantiated are returned when pulling config values (`True` by default).
- `ask_parents` - If `True`, missing keys will check in the parent config node (recursively) before raising a *`SearchFailed`* error (`True` by default).
- `allow_cousins` - (advanced feature) If `True`, missing keys will check in the parent's parent config node (recursively) with the current node's key prepended before raising a *`SearchFailed`* error (`False` by default).

1.16 Saving/Loading Configs

Despite the hierarchical structure of the config object, it can ultimately always be exported as 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 *automatic instantiation* for more details and examples). Consequently, any instantiated objects are **not** included when exporting or saving the config.

To export the config object as a `str`, use `to_yaml()`.

```
import omnifig as fig

cfg = fig.create_config(x=1, y='hello')

print(cfg.to_yaml())

# prints out:
# x: 1
# y: hello
```

Meanwhile, you can save the config file to a file using `export()`.

```
import omnifig as fig

cfg = fig.create_config(x=1, y='hello')

path = cfg.export('my_config', root='/some/path/')

# path is now '/some/path/my_config.yaml'
```

Then the file can be loaded using *`create_config()`* by passing the path to the file in.

```
import omnifig as fig

cfg = fig.create_config('/some/path/my_config.yaml')

print(cfg)

# prints out:
# x: 1
# y: hello
```

1.17 Creators

By default, *pulling values* from the config object will *instantiate* components including *modifiers*. However, you can easily customize how exactly any values are processed by creating custom *creators*.

Creators must be *registered* just like components and modifiers. Then if you want certain components to always use such a custom creator, then you can include that creator name when registering the component.

```
import omnifig as fig

@fig.creator('custom-creator')
class SpecialCreator(fig.Node.DefaultCreator):
    def create_product(self, config, args, kwargs, silent=None):
        # do something special
        print('creating something special')
        return super().create_product(config, args, kwargs, silent=None)

@fig.component('some-component', creator='custom-creator')
class Something(fig.Configurable):
    def __init__(self, x=1):
        self.x = x

# then in a REPL:

>>> cfg = fig.create_config(_type='some-component', x=2)

>>> assert cfg.pull('x') == 2
creating something special

>>> obj = cfg.pull()
creating something special
>>> assert obj.x == 2
>>> assert isinstance(obj, Something)
```

Alternatively, you can specify the creator that should be used directly in the config under the key `_creator`.

```
>>> cfg = fig.create_config(_type='some-component', x=2, _creator='custom-creator')

>>> assert cfg.pull('x') == 2
creating something special
>>> assert obj.x == 2
>>> assert isinstance(obj, Something)
```

Lastly, you can specify the creator as a *setting* of the config object.

```
@fig.component('another-component') # without specifying a creator
class Simple(fig.Configurable):
    def __init__(self, y=1):
        self.y = y

# then in a REPL:

>>> cfg = fig.create_config(_type='another-component', y=5)

>>> obj1 = cfg.create() # uses default creator
>>> assert obj1.y == 5
>>> assert isinstance(obj1, Simple)

>>> with cfg.context(creator='custom-creator'):
...     obj2 = cfg.create() # now uses the custom creator
creating something special
>>> assert obj2.y == 5
>>> assert isinstance(obj2, Simple)

>>> obj3 = cfg.create() # outside of the context, using default creator
>>> assert obj3.y == 5
>>> assert isinstance(obj3, Simple)

>>> cfg.settings['creator'] = 'custom-creator'

>>> obj4 = cfg.create() # now back to custom-creator
creating something special
>>> assert obj4.y == 5
>>> assert isinstance(obj4, Simple)
```

1.18 Abstract Base Classes

class omnifig.abstract.**AbstractConfig**

Bases: `object`

Abstract class for config objects

exception SearchFailed

Bases: `KeyError`

Raised when a search fails

property cro: Tuple[str, ...]

Returns the list of all config files that were composed to produce this config tree. Analogous to the method resolution order (`mro`) for classes.

Return type
`Tuple[str, ...]`

property bases: Tuple[str, ...]

Returns the list of config files that were explicitly mentioned to produce this config tree. Analogous to `__bases__` for classes.

Return type
Tuple[str, ...]

property project: *AbstractProject*

Returns the project object associated with this config

Return type
AbstractProject

property root: *AbstractConfig*

Returns the root node of the config object

Return type
AbstractConfig

export(*name*, *, *root=None*, *fmt=None*)

Exports the config object to the given path.

Parameters

- **name** (Union[str, Path]) – Name/path to export the config object to.
- **root** (Union[str, Path, None]) – Path to use as the root of the config object (defaults to the current working directory).
- **fmt** (Optional[str]) – Format to export the config object in (if None, will be inferred from the path).

Return type
Optional[Path]

peek(*query=None*, *default=<class 'omnibelt.typing.unspecified_argument'>*, *, *silent=None*)

Returns a config object given by searching for the query in the config object.

Parameters

- **query** (Optional[str]) – The query to search for. If None, returns self.
- **default** (Optional[Any]) – The default value to return if the query is not found.
- **silent** (Optional[bool]) – If True, no message is reported based on the search.

Return type
AbstractConfig

Returns
The config object found by the query, or the default value if the query is not found, and returns self if no query is provided.

pull(*query=None*, *default=<class 'omnibelt.typing.unspecified_argument'>*, *, *silent=None*)

Returns the value found by searching for the query in the config object.

Parameters

- **query** (Optional[str]) – The query to search for. If None, returns default.
- **default** (Optional[Any]) – The default value to return if the query is not found.
- **silent** (Optional[bool]) – If True, no message is reported based on the search.

Return type
Any

Returns
The value found by the query, or the default value if the query is not found, and returns the value of self if no query is provided.

push_pull(*addr*, *value*, *, *overwrite=True*, *silent=None*)

Composes the push and pull methods into a single method. Can be used to set a value if none exists, and otherwise pull the existing value.

Parameters

- **addr** (str) – The address to push/pull from.
- **value** (Any) – The value to push to the address.
- **overwrite** (bool) – If True, the value is pushed to the address even if it already exists.
- **silent** (Optional[bool]) – If True, no message is reported.

Return type
Any

Returns
The value found at the address.

push_peek(*addr*, *value*, *, *overwrite=True*, *silent=None*)

Composes the push and peek methods into a single method. Can be used to set a value if none exists, and otherwise peek the existing value.

Parameters

- **addr** (str) – The address to push/peek from.
- **value** (Any) – The value to push to the address.
- **overwrite** (bool) – If True, the value is pushed to the address even if it already exists.
- **silent** (Optional[bool]) – If True, no message is reported.

Return type
AbstractConfig

Returns
The value found at the address.

peeks(**queries*, *default=<class 'omnibelt.typing.unspecified_argument'>*, *silent=None*)

Returns a config object given by searching for the queries in the config object. If multiple queries are provided, each query is searched if the previous query fails.

Parameters

- ***queries** – Any number of queries to search for.
- **default** (Optional[Any]) – The default value to return if none of the queries are not found.
- **silent** (Optional[bool]) – If True, no message is reported based on the search.

Return type
AbstractConfig

Returns
The config object found by the query, or the default value if none of the queries are found.

pulls(**queries*, *default=<class 'omnibelt.typing.unspecified_argument'>*, *silent=None*)

Returns the value found by searching for the queries in the config object.

Parameters

- ***queries** (str) – Any number of queries to search for.
- **default** (Optional[Any]) – The default value to return if none of the queries are not found.
- **silent** (Optional[bool]) – If True, no message is reported based on the search.

Return type
Any

Returns
The value found by the query, or the default value if none of the queries are found.

push(*addr*, *value*, *overwrite=True*, ***, *silent=None*)

Sets the value at the given address.

Parameters

- **addr** (str) – The address to set the value at.
- **value** (Any) – The value to set.
- **overwrite** (bool) – If True, the value is set even if it already exists.
- **silent** (Optional[bool]) – If True, no message is reported.

Return type
bool

Returns
True if the value was set, False if the value was not set.

update(*update*)

Updates the config object with the given config object (recursively overwriting common keys).

Parameters
update (*AbstractConfig*) – Config object to update with.

Return type
AbstractConfig

Returns
The updated config object (self).

silence(*silent=True*)

Context manager to silence the config object.

Parameters
silent (bool) – If True, no messages are reported when querying the config object.

Return type
ContextManager

Returns
Context manager to silence the config object.

peek_named_children(***, *silent=None*)

Returns an iterator of the child config objects of self together with their keys.

Parameters
silent (Optional[bool]) – If True, no messages are reported.

Return type
Iterator[Tuple[str, *AbstractConfig*]]

Returns
An iterator of the child config objects of self together with their keys.

peek_children(*, *silent=None*)

Returns an iterator of the child config objects of self.

Parameters
silent (Optional[bool]) – If True, no messages are reported.

Return type
Iterator[*AbstractConfig*]

Returns
An iterator of the child config objects of self.

pull_named_children(*, *force_create=False*, *silent=None*)

Returns an iterator of the child values of self together with their keys.

Parameters

- **force_create** (Optional[bool]) – If True, creates new child values even if they already exist.
- **silent** (Optional[bool]) – If True, no messages are reported.

Return type
Iterator[Tuple[str, Any]]

Returns
An iterator of the child values of self together with their keys.

pull_children(*, *force_create=False*, *silent=None*)

Returns an iterator of the child values of self.

Parameters

- **force_create** (Optional[bool]) – If True, creates new child values even if they already exist.
- **silent** (Optional[bool]) – If True, no messages are reported.

Return type
Iterator[Any]

Returns
An iterator of the child values of self.

create(**args*, ***kwargs*)

Creates a new value based on the contents of self.

Parameters

- ***args** (Any) – Manual arguments to pass to the value constructor.
- ****kwargs** (Any) – Manual keyword arguments to pass to the value constructor.

Return type
Any

Returns
The newly created value.

`create_silent`(**args*, ***kwargs*)

Creates a new value based on the contents of self silently.

Parameters

- **`*args`** (`Any`) – Manual arguments to pass to the value constructor.
- **`**kwargs`** (`Any`) – Manual keyword arguments to pass to the value constructor.

Return type
`Any`

Returns
The newly created value.

`peek_create`(*query*, *default=<class 'omnibelt.typing.unspecified_argument'>*, **args*, ***kwargs*)

Composes the peek and create methods into a single method.

Can be used to create a value using the config object found with the query.

Parameters

- **`query`** – The query to search for.
- **`default`** (`Optional[Any]`) – If provided, the value is created using the default config object.
- **`*args`** – Manual arguments to pass to the value constructor.
- **`**kwargs`** – Manual keyword arguments to pass to the value constructor.

Returns
The newly created value or *default* if the query is not found.

`process`(**args*, ***kwargs*)

Processes the config object using the contents of self.

If a value for this config object has already been created, it is returned instead of creating a new one.

Parameters

- **`*args`** (`Any`) – Manual arguments to pass to the value constructor. (ignored if a value has already been created)
- **`**kwargs`** (`Any`) – Manual keyword arguments to pass to the value constructor.
- **`created)`** (*`(ignored if a value has already been`*) –

Return type
`Any`

Returns
The processed value.

`peek_process`(*query*, *default=<class 'omnibelt.typing.unspecified_argument'>*, **args*, ***kwargs*)

Composes the peek and process methods into a single method.

Parameters

- **`query`** – The query to search for.
- **`default`** (`Optional[Any]`) – If provided, the value is processed using the default config object.

- ***args** – Manual arguments to pass to the value constructor. (ignored if a value has already been created)
- ****kwargs** – Manual keyword arguments to pass to the value constructor.
- **created)** (*(ignored if a value has already been*) –

Returns
The processed value or *default* if the query is not found.

class omnifig.abstract.**AbstractConfigurable**

Bases: `object`

Abstract mix-in for objects that can be constructed using a config object.

classmethod init_from_config(*config*, *args=None*, *kwargs=None*, *, *silent=None*)

Constructor to initialize a class informed by the config object *config*.

It is recommended that this should be a parent of any class that is registered as a component or modifier.

Parameters

- **config** (*AbstractConfig*) – Config object to fill in any needed parameters.
- **args** (`Tuple`) – Manually specified arguments to pass to the constructor.
- **kwargs** (`Dict[str, Any]`) – Manually specified keyword arguments to pass to the constructor.
- **silent** (`Optional[bool]`) – If True, no messages are reported when querying the config object.

Return type
AbstractConfigurable

Returns
Initialized instance of this class.

class omnifig.abstract.**AbstractCertifiable**

Bases: *AbstractConfigurable*

Abstract mix-in for objects that can must be certified after intialization.

class omnifig.abstract.**AbstractConfigManager**

Bases: `object`

Abstract class for config managers.

exception ConfigNotRegistered

Bases: `KeyError`

A config was not registered

ConfigNode = None

register_config(*name*, *path*)

Registers a path as a config file with the given name.

Parameters

- **name** (`str`) – Name to register the config file under.
- **path** (`Union[str, Path]`) – Path to the config file.

Return type
NamedTuple

Returns
The entry that was registered (should contain at least the attributes for the name and path).

`register_config_dir`(*root*)

Recursively registers all config files found in the given directory.

Parameters
root (Union[str, Path]) – Path to the directory of the config files that should be registered.

Return type
List[NamedTuple]

Returns
A list of the names of the registered config files.

`iterate_configs`()

Iterates over all registered config files.

Return type
Iterator[NamedTuple]

Returns
An iterator over all registered config files.

`find_local_config_entry`(*name*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the entry for the config file with the given name in the registry.

Parameters

- **name** (str) – Name of the config file to find.
- **default** (Optional[Any]) – Default value to return if the artifact is not found.

Return type
NamedTuple

Returns
The entry for the config file with the given name.

Raises
KeyError – If the config file is not registered.

`find_project_config_entry`(*name*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the entry for a config by name. Including checking the nonlocal projects.

Parameters

- **name** (str) – of the config file used to register the config
- **default** (Optional[Any]) – Default value to return if the artifact is not found.

Return type
NamedTuple

Returns
The entry for the config

Raises
ConfigNotFoundError – if the config is not registered

find_config_path(*name*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the path to the config file with the given name in the registry.

Parameters

- **name** (str) – Name of the config file to find.
- **default** (Optional[Any]) – Default value to return if the artifact is not found.

Return type
Path

Returns
The path to the config file with the given name.

Raises
KeyError – If the config file is not registered.

parse_argv(*argv*, *script_name=<class 'omnibelt.typing.unspecified_argument'>*)

Parses the given arguments and returns a config object.

Arguments are expected in the following order (all of which are optional):

1. Meta rules to modify the config loading process and run mode.
2. Name of the script to run.
3. Names of registered config files that should be loaded and merged (in order of precedence).
4. Manual config parameters (usually keys, prefixed by `--` and corresponding values)

Parameters

- **argv** (Sequence[str]) – List of arguments to parse (expected to be `sys.argv[1:]`).
- **script_name** (Optional[str]) – Manually specified name of the script (defaults to what is specified in the resulting config).

Return type
AbstractConfig

Returns
Config object containing the parsed arguments.

create_config(*configs=None*, *data=None*)

Creates a config object from the given config file names and provided arguments.

Parameters

- **configs** (Optional[Sequence[str]]) – Names of registered config files to load and merge (in order of precedence).
- **data** (Union[Dict[str, Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], List[Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], str, int, float, bool, None]) – Manual config parameters to populate the config object with.

Return type
AbstractConfig

Returns
Config object resulting from loading/merging *configs* and including *data*.

`load_raw_config`(*path*)

Loads a config file from the given path and returns the raw config data (made up of standard python objects, such as `dict` and `list`).

Parameters
path (`Union`[`str`, `Path`]) – Path to the config file to load.

Return type
`Union`[`Dict`[`str`, `Union`[`Dict`[`str`, JSONABLE], `List`[JSONABLE], `str`, `int`, `float`, `bool`, `None`]], `List`[`Union`[`Dict`[`str`, JSONABLE], `List`[JSONABLE], `str`, `int`, `float`, `bool`, `None`]], `str`, `int`, `float`, `bool`, `None`]

Returns
Config data loaded from the given path.

Raises

- **FileNotFoundError** – If the config file does not exist.
- **ValueError** – If the config file cannot be loaded.

`configurize`(*raw*)

Converts the given raw config data into a config object.

Raw config data can include primitives, lists, and dicts (including OrderedDict or tuples), but should not include any other types.

Parameters
raw (`Union`[`Dict`[`str`, `Union`[`Dict`[`str`, JSONABLE], `List`[JSONABLE], `str`, `int`, `float`, `bool`, `None`]], `List`[`Union`[`Dict`[`str`, JSONABLE], `List`[JSONABLE], `str`, `int`, `float`, `bool`, `None`]], `str`, `int`, `float`, `bool`, `None`]) – Raw config data to convert into a config object.

Return type
AbstractConfig

Returns
Config object created from the given raw config data.

`merge_configs`(**configs*)

Merges the given config objects into a single config object.

Including recursively merging any nested config objects and overwriting any duplicate keys in reverse provided order (so the configs should be provided in order of precedence).

Parameters
***configs** (*AbstractConfig*) – Provided config objects to merge.

Return type
AbstractConfig

Returns
Config object resulting from merging the given config objects.

`static update_config`(*base*, *update*)

Updates the config object with the given config object (recursively overwriting common keys).

Parameters

- **base** (*AbstractConfig*) – Config object to update.
- **update** (*AbstractConfig*) – Config object to update *base* with.

Return type
AbstractConfig

Returns
The updated config object *base*.

class omnifig.abstract.**AbstractCustomArtifact**

Bases: object

static top(*config*, **args*, ***kwargs*)

Return type
Any

get_wrapped()

Return type
Union[Callable, Type]

class omnifig.abstract.**AbstractCreator**(*config*, ***kwargs*)

Bases: object

Abstract class for creators.

classmethod replace(*creator*, *config*, ***kwargs*)

Extracts any required information from the required *creator* to create and return a new creator.

Parameters

- **creator** (*AbstractCreator*) – base creator to extract information from.
- **config** (*AbstractConfig*) – config object to initialize the new creator with.
- ****kwargs** – other keyword arguments to pass to the new creator initialization.

Return type
AbstractCreator

Returns
New creator created from the given *creator* and *config*.

__init__(*config*, ***kwargs*)

create(*config*, **args*, ***kwargs*)

Creates an object from the given config node and other arguments.

Parameters

- **config** (*AbstractConfig*) – Config node to create the object from.
- ***args** (Any) – Manual arguments to pass to the object.
- ****kwargs** (Any) – Manual keyword arguments to pass to the object.

Return type
Any

Returns
Object created from the given config node and other arguments.

create_product(*config*, *args=None*, *kwargs=None*)

Creates an object from the given config node and other arguments.

Parameters

- **config** (*AbstractConfig*) – Config node to create the object from.
- **args** (Optional[Tuple]) – Manual arguments to pass to the object.
- **kwargs** (Optional[Dict[str, Any]]) – Manual keyword arguments to pass to the object.

Return type
Any

Returns
Object created from the given config node and other arguments.

class omnifig.abstract.**AbstractRunMode**(**args*, ***kwargs*)

Bases: *Activatable*

Abstract class for run modes. Run modes include Projects and Profiles

main(*argv*, *script_name=None*)

Runs the script with the given arguments using the config object obtained by parsing `argv`.

Parameters

- **argv** (Sequence[str]) – List of top-level arguments (expected to be `sys.argv[1:]`).
- **script_name** (Optional[str]) – specified name of the script
- **object).** ((*defaults to what is specified in argv when it is parsed into a config*) –

Return type
Any

Returns
The output of the script.

run_script(*script_name*, *config*, **args*, ***kwargs*)

Runs the script with the given arguments using *run()* of the current project.

Parameters

- **script_name** (str) – The script name to run (must be registered).
- **config** (*AbstractConfig*) – Config object to run the script with (must include the script under `_meta.script_name`).
- ***args** (Any) – Manual arguments to pass to the script.
- ****kwargs** (Any) – Manual keyword arguments to pass to the script.

Return type
Any

Returns
The output of the script.

run(*config*, **args*, ***kwargs*)

Runs a script with the given config object and other arguments.

Before running the script using `run_local()`, the config object is validated with *validate_run()*, which can modify the run mode.

Parameters

- **config** (*AbstractConfig*) – Config object to run the script with (must include the script under `_meta.script_name`).

- **args** (Any) – Manual arguments to pass to the script.
- **kwargs** (Any) – Manual keyword arguments to pass to the script.

Return type
Any

Returns
The output of the script.

cleanup()

After running the script through *main()*, this method is called to clean up any resources used by the run mode.

Returns
None

behaviors()

Iterates over all behaviors associated with this project.

Return type
Iterator[*AbstractBehavior*]

validate_run(*config*)

Validates the config object before running the script.

Parameters
config (*AbstractConfig*) – Config object to validate.

Return type
Optional[*AbstractRunMode*]

Returns
None if the config is valid, otherwise the run mode to transfer to.

parse_argv(*argv*, *, *script_name=None*)

Parses the given arguments and returns a config object.

Arguments are expected in the following order (all of which are optional):

1. Meta rules to modify the config loading process and run mode.
2. Name of the script to run.
3. Names of registered config files that should be loaded and merged (in order of precedence).
4. Manual config parameters (usually keys, prefixed by `--` and corresponding values)

Parameters

- **argv** (Sequence[str]) – List of arguments to parse (expected to be `sys.argv[1:]`).
- **script_name** (Optional[str]) – Manually specified name of the script (defaults to what is specified in the resulting config).

Return type
AbstractConfig

Returns
Config object containing the parsed arguments.

class omnifig.abstract.**AbstractProject**(*path=None*, *profile=None*, ***kwargs*)

Bases: *AbstractRunMode*, *FileInfo*, *Activatable*

Abstract class for projects. Projects track artifacts (eg. configs and components) in registries and manage loading configs and running scripts.

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

Loads the info if the provided *data* is a file path.

Parameters

- **data** – A file path to a yaml file, or a dictionary containing the info
- ****kwargs** – Other arguments passed on to `super()`

property **profile**: *AbstractProfile*

Profile object that the project is associated with.

Return type
AbstractProfile

Returns
Profile object.

nonlocal_projects()

Iterator over all projects that are related to this one, followed by all active base projects of the profile (without repeating any projects).

Return type
`Iterator[NamedTuple]`

Returns
An iterator over all projects that are related to this one, followed by all active base projects

behaviors()

Iterates over all behavior instances associated with this project.

By default, this creates an instance for all behaviors in the profile.

Return type
`Iterator[NamedTuple]`

Returns
Iterator over meta rules in order of priority (highest -> least).

exception **UnknownArtifactError**(*artifact_type*, *ident*)

Bases: `NotFoundError`

Raised when trying to find an artifact that is not registered.

__init__(*artifact_type*, *ident*)

xray(*artifact*, ***, *sort=False*, *reverse=False*, *as_list=False*)

Prints a list of all artifacts of the given type accessible from this project (including related and active base projects).

Parameters

- **artifact** (`str`) – artifact type (e.g. 'script', 'config')
- **sort** (`Optional[bool]`) – sort the list of artifacts by name
- **reverse** (`Optional[bool]`) – reverse the order of the list of artifacts

- **as_list** (Optional[bool]) – instead of printing, return the list of artifacts

Returns
if as_list is True, returns a list of artifacts

Return type
list

Raises
UnknownArtifactTypeError – if the given artifact type does not exist

find_local_artifact(*artifact_type*, *ident*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the artifact with the given type and identifier in `self` without checking related projects or active projects in the profile.

Parameters

- **artifact_type** (str) – The type of artifact to find.
- **ident** (str) – The identifier of the artifact to find.
- **default** (Optional[Any]) – The default value to return if the artifact is not found.

Return type
Optional[NamedTuple]

Returns
The artifact entry from the registry corresponding to the given type.

Raises

- *UnknownArtifactTypeError* – If the artifact type is not registered.
- *UnknownArtifactError* – If the artifact is not found and no default is given.

find_artifact(*artifact_type*, *ident*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds an artifact in the project's registries. Artifacts are data or functionality such as configs and components.

Parameters

- **artifact_type** (str) – Type of artifact to find (eg. 'config' or 'component').
- **ident** (str) – Name of the artifact that was registered.
- **default** (Optional[Any]) – Default value to return if the artifact is not found.

Return type
NamedTuple

Returns
Artifact object, or, if a default value is given and artifact is not found.

Raises
UnknownArtifactError – if the artifact is not found and no default is specified.

register_artifact(*artifact_type*, *ident*, *artifact*, ***kwargs*)

Registers a new artifact in the project's registries. Overwrites any existing artifact with the same name.

Parameters

- **artifact_type** (str) – Type of artifact to register (eg. 'config' or 'component').
- **ident** (str) – Name of the artifact to be registered.

- **artifact** (Union[Type, Callable]) – Artifact object to register (usually a function, type, or path).
- ****kwargs** (Any) – Optional additional parameters to store with the artifact.

Return type
NamedTuple

Returns
Registration entry for the artifact.

iterate_artifacts(*artifact_type*)

Iterates over all artifacts of the given type in the project's registries.

Parameters
artifact_type (str) – Type of artifact to iterate (eg. 'config' or 'component').

Return type
Iterator[NamedTuple]

Returns
Iterator over all artifacts of the given type.

create_config(**configs*, ***parameters*)

Creates a new config object with the given parameters.Creates a config object from the given config file names and provided parameters.

Parameters

- ***configs** (str) – Names of registered config files to load and merge (in order of precedence).
- ****parameters** (Union[str, int, float, bool, None]) – Manual config parameters to populate the config object with.

Return type
AbstractConfig

Returns
Config object.

quick_run(*script_name*, **configs*, ***parameters*)

Composes create_config() and :func:`run_script() into a single method to first create a config object and then run the specified script.

Parameters

- **script_name** (str) – Name of the script to run (should be registered).
- ***configs** (str) – Names of registered config files to load and merge (in order of precedence).
- ****parameters** (Union[Dict[str, Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], List[Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], str, int, float, bool, None]) – Manual config parameters to populate the config object with.

Return type
Any

Returns
Output of the script.

class omnifig.abstract.**AbstractProfile**(*data=None*, ***kwargs*)

Bases: *FileInfo*, *Activatable*

Abstract classes for profiles. Profiles manage projects and provide meta rules. Unlike projects, generally a runtime should only use a single global instance of a profile.

classmethod get_project_type(*ident*)

Gets the project type entry for the given identifier (from a registry).

Parameters
ident (str) – Name of the registered project type.

Return type
NamedTuple

Returns
Project type entry.

classmethod replace_profile(*profile*)

Replaces the current profile instance with the given profile. This is used to set the global profile.

Parameters
profile (*AbstractProfile*) – New profile instance.

Return type
AbstractProfile

Returns
Old profile instance (which is now replaced).

classmethod get_profile()

Gets the current profile instance of the runtime environment.

Return type
AbstractProfile

Returns
Profile instance.

classmethod register_behavior(*name*, *typ*, ***, *description=None*)

Registers a new behavior in the profile.

Behaviors are classes which are instantiated and managed by .

Parameters

- **name** (str) – Name of the behavior.
- **typ** (Type[*AbstractBehavior*]) – Behavior class (recommended to subclass *AbstractBehavior*).
- **description** (Optional[str]) – Description of the behavior.

Return type
NamedTuple

Returns
Registration entry for the behavior.

classmethod get_behavior(*name*)

Gets the behavior entry for the given identifier (from the registry).

Parameters
name (str) – Name of the registered behavior.

Return type
NamedTuple

Returns
Behavior entry.

classmethod iterate_behaviors()

Iterates over all registered behaviors.

Return type
Iterator[NamedTuple]

Returns
Iterator over all behavior entries.

entry(*script_name=None*)

Primary entry point for the profile. This method is called when using the **fig** command.

Parameters
script_name (Optional[str]) – Manually specified script name to run (if not provided, will be parsed from sys.argv).

Return type
None

Returns
None

initialize(**projects*)

Initializes the specified projects (including activating them, which generally registers all associated configs and imports files and packages)

Parameters
***projects** (str) – Identifiers of projects to initialize (activates the current project only, if none is provided).

Return type
None

Returns
None

main(*argv*, ***, *script_name=None*)

Runs the script with the given arguments using *main()* of the current project.

Parameters

- **argv** (Sequence[str]) – List of top-level arguments (expected to be sys.argv[1:]).
- **script_name** (Optional[str]) – specified name of the script
- **object).** *((defaults to what is specified in argv when it is parsed into a config)* –

Return type
Any

Returns
The output of the script.

run_script(*script_name*, *config*, **args*, ***kwargs*)

Runs the script registered with the given name and the given arguments using *run_script()* of the current project.

Parameters

- **script_name** (str) – Name of the script to run (must be registered).
- **config** (*AbstractConfig*) – Config object to run the script with.
- ***args** (Any) – Manual arguments to pass to the script.
- ****kwargs** (Any) – Manual keyword arguments to pass to the script.

Return type
Any

Returns
The output of the script.

run(*config*, *, *args=None*, *kwargs=None*)

Runs the script with the given arguments using *run()* of the current project.

This method assumes the script_name is already contained in the config, otherwise use *run_script()*.

Parameters

- **config** (*AbstractConfig*) – Config object to run the script with (must include the script under _meta.script_name).
- **args** (Optional[Tuple]) – Manual arguments to pass to the script.
- **kwargs** (Optional[Dict[str, Any]]) – Manual keyword arguments to pass to the script.

Return type
Any

Returns
The output of the script.

quick_run(*script_name*, **configs*, ***parameters*)

Creates a config object and runs the script using *quick_run()* of the current project.

Parameters

- **script_name** (str) – Name of the script to run (must be registered).
- ***configs** (str) – Names of registered config files to load and merge (in order of precedence).
- ****parameters** (Any) – Manual config parameters to populate the config object with.

Return type
Any

Returns
Output of the script.

cleanup()

Calls *cleanup()* of the current project. Generally not needed to be called manually.

Return type
None

Returns
None

`create_config`(**configs*, ***parameters*)

Process the provided data to create a config object (using the current project).

Parameters

- **configs** (str) – usually a list of parent configs to be merged
- **parameters** (Union[Dict[str, Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], List[Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], str, int, float, bool, None]) – any manual parameters to include in the config object

Return type
AbstractConfig

Returns
Config object resulting from loading/merging *configs* and including *data*.

`parse_argv`(*argv*, *script_name=None*)

Parses the given arguments and returns a config object.

Arguments are expected in the following order (all of which are optional):

1. Meta rules to modify the config loading process and run mode.
2. Name of the script to run.
3. Names of registered config files that should be loaded and merged (in order of precedence).
4. Manual config parameters (usually keys, prefixed by `--` and corresponding values)

Parameters

- **argv** (Sequence[str]) – List of arguments to parse (expected to be `sys.argv[1:]`).
- **script_name** – Manually specified name of the script (defaults to what is specified in the resulting config).

Return type
AbstractConfig

Returns
Config object containing the parsed arguments.

`extract_info`(*other*)

Extract data from the provided profile instance and store it in self.

Recommended to use if a project expects a custom profile different from the currently used one.

Parameters
profile – Base profile instance.

Return type
None

Returns
None

`get_current_project`()

Gets the current project instance.

Return type
AbstractProject

Returns
Current project instance.

`switch_project`(*ident=None*)

Switches the current project to the one with the given identifier.

Parameters
ident (`Optional[str]`) – Name of the project to switch to, defaults to the default project (with name: None).

Return type
AbstractProject

Returns
New current project instance.

`project_context`(*ident=None*)

Context manager to temporarily switch to a different current project.

Parameters
ident (`Union[str, AbstractProject, None]`) – Name of the project to switch to, defaults to the default project (with name: None).

Return type
`ContextManager[AbstractProject]`

Returns
Context manager to switch to the specified project.

`iterate_projects`()

Iterates over all loaded projects.

Return type
`Iterator[AbstractProject]`

Returns
Iterator over all loaded project instances.

`get_project`(*ident=None*)

Gets the project with the given identifier, if the project is not already loaded, it will be loaded.

Parameters
ident (`Optional[str]`) – Name of the project to get, defaults to the default project (with name: None).

Return type
AbstractProject

Returns
Project instance.

`class omnifig.abstract.AbstractBehavior`(*project*, ***kwargs*)

Bases: `object`

Interface for meta rules.

`__init__`(*project*, ***kwargs*)

Behaviors are usually instantiated by the project at the beginning `main()` method.

Otherwise, if `main()` is not used, the behaviors are instantiated in `run()` right before the behaviors are filtered using *include()*.

Parameters

- **project** (*AbstractProject*) – Project of this behavior instance.
- ****kwargs** (Any) – Additional keyword arguments (unused).

exception TerminationFlag(*out=None*)

Bases: KeyboardInterrupt

Raised if the subsequent script should not be run.

__init__(*out=None*)

Prevents the subsequent script from being run.

Parameters

out (Any) – User-defined output to be returned instead of the output of the script.

out = None

static parse_argv(*meta*, *argv*, *script_name=None*)

Optionally modifies the arguments when the project's main() is called.

Parameters

- **meta** (Dict[str, Any]) – Meta-data extracted from the argv so far (can be modified here).
- **argv** (List[str]) – List of arguments to parse (expected to be sys.argv[1:]).
- **script_name** (Optional[str]) – Manually specified name of the script (if not provided, it will be parsed from argv).

Return type

Optional[List[str]]

Returns

Modified list of arguments (or None if no modification is needed).

static validate_project(*config*)

Validates the project (provided in constructor) using the given config object before running it.

If a different project should be used, it can be returned here.

Parameters

config (*AbstractConfig*) – The config object to use for validation.

Return type

Optional[*AbstractProject*]

Returns

The new project to use for the script execution or None if no new project was returned.

static include(*meta*)

Checks if the current behavior should be included in the subsequent script execution.

Parameters

meta (*AbstractConfig*) – Meta config to configure behavior and script execution.

Return type

bool

Returns

True if the behavior should be included, False otherwise.

static pre_run(*meta*, *config*)

Runs before the script is executed, within the project run().

Parameters

- **meta** (*AbstractConfig*) – Meta config to configure behavior and script execution.
- **config** (*AbstractConfig*) – Config object which will be passed to the script.

Return type
Optional[*AbstractConfig*]

Returns
New config object to pass to the script instead (or None to use the original config).

Raises
TerminationFlag – If the subsequent script should not be run.

exception IgnoreException(*out=None*)

Bases: Exception

Raised if the underlying exception raised while running the script should be ignored.

__init__(*out=None*)

Instead of raising the exception, out will be returned.

Parameters
out (Any) – User-defined output to be returned instead of raising the exception.

static handle_exception(*meta*, *config*, *exc*)

Runs if the script raises an exception.

Parameters

- **meta** (*AbstractConfig*) – Meta config to configure behavior and script execution.
- **config** (*AbstractConfig*) – Config object to run the script with.
- **exc** (Exception) – Exception that was raised.

Return type
None

Returns
None

Raises

- ***IgnoreException*** – If the exception should be ignored.
- ***TerminationFlag*** – If the subsequent behaviors should not be run.

static post_run(*meta*, *config*, *output*)

Function called after the script is run. If the function returns a value, it will be used as the output of the script.

Parameters

- **meta** (*AbstractConfig*) – Meta config to configure behavior and script execution.
- **config** (*AbstractConfig*) – Config object used to run the script with.
- **output** (Any) – Output of the script.

Return type
Optional[Any]

Returns
New output of the script, or None to use the original output.

Raises
TerminationFlag – If the subsequent behaviors should not be run.

static cleanup()

Function called at the end of the project `main()` during cleanup.

Note that this is only called if the project `main()`. To define a cleanup function for the behavior that is called every time a script is run, use *post_run()*.

Returns
None

class omnifig.config.abstract.**AbstractSearch**(*origin*, *queries*, *default*, ***kwargs*)

Bases: `object`

Abstract class for search objects used by the config object to find the format data

__init__(*origin*, *queries*, *default*, ***kwargs*)

exception SearchFailed(**queries*)

Bases: *SearchFailed*

Raised when a search fails to find a value

__init__(**queries*)

find_node(*silent=None*)

Finds the node that contains the product

Return type
AbstractConfig

find_product(*silent=None*)

Finds the node that contains the product, and then extracts the product

Return type
Any

static sub_search(*origin*)

Creates a context manager for new searches to be able to reference the original search *origin*

Return type
ContextManager

class omnifig.config.abstract.**AbstractReporter**

Bases: `object`

Abstract class for reporters used by the config object to report changes

static log(**msg*, *end='\\n'*, *sep=' '*, *silent=None*)

Prints the given message to the console

Return type
str

get_key(*trace*)

Returns the key of the node resulting of the search

Return type
str

report_node(*node*, *, *silent=None*)

Reports information about a config node

Return type
Optional[str]

report_product(*node*, *, *silent=None*)

Reports the product of a config node

Return type
Optional[str]

report_default(*node*, *default*, *, *silent=None*)

Reports a config node defaulted to the given value

Return type
Optional[str]

report_iterator(*node*, *product=False*, *, *silent=None*)

Reports the start of an iterator over the config node

Return type
Optional[str]

reuse_product(*node*, *product*, *, *silent=None*)

Reports that the product of the given node is being reused

Return type
Optional[str]

create_primitive(*node*, *value=<class 'omnibelt.typing.unspecified_argument'>*, *, *silent=None*)

Reports that the product of the given node is a primitive

Return type
Optional[str]

create_container(*node*, *, *silent=None*)

Reports that the product of the given node is a container (e.g. a dict or list)

Return type
Optional[str]

create_component(*node*, *, *component_type=None*, *modifiers=None*, *creator_type=None*, *silent=None*)

Reports that the product of the given node is a component

Return type
Optional[str]

1.19 Behaviors

class omnifig.behaviors.base.**Behavior**(*project*, ***kwargs*)

Bases: *AbstractBehavior*

Recommended parent class for meta rules.

name = None

code = None

priority = 0

`num_args = 0`

`description = None`

`parse_argv`(*meta*, *argv*, *script_name=None*)

Optionally modifies the arguments when the project's `main()` is called.

Parameters

- **meta** (`Dict[str, Any]`) – Meta-data extracted from the argv so far (can be modified here).
- **argv** (`List[str]`) – List of arguments to parse (expected to be `sys.argv[1:]`).
- **script_name** (`Optional[str]`) – Manually specified name of the script (if not provided, it will be parsed from argv).

Return type
`Optional[List[str]]`

Returns
Modified list of arguments (or `None` if no modification is needed).

`include`(*meta*)

Checks if the current behavior should be included before running the given config.

Parameters
config – Config object to use.

Return type
`bool`

Returns
`True` if the behavior should be included, `False` otherwise.

`class omnifig.behaviors.help.Help`(*project*, ***kwargs*)

Bases: *Behavior*

When activated, this behavior prints the help message for the current project (and then exits the program).

`classmethod format_selected_script`(*config*, *entry*, *verbose=False*)

Formats the help message about the selected script.

Return type
`str`

`classmethod format_scripts`(*config*, *entries*, *verbose=False*)

Formats the help message about the available scripts.

Return type
`str`

`classmethod format_behaviors`(*config*, *entries*, *verbose=False*)

Formats the help message about the available behaviors.

Return type
`str`

`static format_configs`(*config*, *entries*, *verbose=False*)

Formats the help message about the available configs.

Return type
`str`

classmethod pre_run(*meta*, *config*)

When activated, this behavior prints the help message for the current project (and then exits the program).

Parameters

- **meta** (*AbstractConfig*) – The meta config object (used to check if the behavior is activated with the `quiet` key)
- **config** (*AbstractConfig*) – The config object to be modified

Return type
`None`

Returns
`None`

Raises
TerminationFlag – if the rule is activated, to prevent the script from running

code = 'h'

description = 'Display this help message'

name = 'help'

num_args = 0

priority = 99

class omnifig.behaviors.debug.**Debug**(*project*, ***kwargs*)

Bases: *Behavior*

When activated, this behavior updates the config object with the config file `debug`. If for any reason the config file `debug` has already been loaded, this behavior will do nothing.

Note that only a local `debug` config is merged, so the `debug` config file must be registered with the current project.

__init__(*project*, ***kwargs*)

Sets up the debug behavior, including setting up a flag to make sure the debug config is only loaded once.

Parameters

- **project** (*AbstractProject*) – used to create the debug config
- ****kwargs** (`Any`) – passed to super

pre_run(*meta*, *config*)

When activated, this behavior updates the config object with the config file `debug`.

If for any reason the config file `debug` has already been loaded, this behavior will do nothing.

Parameters

- **meta** (*AbstractConfig*) – The meta config object (used to check if the behavior is activated with the `debug` key)
- **config** (*AbstractConfig*) – Config object which will be passed to the script.

Return type
`Optional[AbstractConfig]`

Returns
the config object possibly updated with the `debug` config file

Raises
ConfigNotFoundError – if the config file debug does not exist

code = 'd'

description = 'Switch to debug mode'

name = 'debug'

num_args = 0

priority = 100

class omnifig.behaviors.quiet.**Quiet**(*project*, ***kwargs*)

Bases: *Behavior*

When activated, this behavior sets the config object to silent mode, which means pulls/pushes are not printed to stdout

__init__(*project*, ***kwargs*)

Sets the attribute to keep track of what the previous value of `config.silent` was

pre_run(*meta*, *config*)

When activated, this will set the config object to silent mode

Parameters

- **meta** (*AbstractConfig*) – The meta config object (used to check if the rule is activated with the `quiet` key)
- **config** (*AbstractConfig*) – The config object to be modified

Return type
None

Returns
None

post_run(*meta*, *config*, *output*)

When activated, this rule sets the config back to its previous value

Parameters

- **meta** (*AbstractConfig*) – The meta config object (not used)
- **config** (*AbstractConfig*) – Config object used to run the script.
- **output** (Any) – Output of the script.

Return type
None

Returns
None

code = 'q'

description = 'Set config to silent'

name = 'quiet'

num_args = 0

priority = 10

1.20 Config Objects

class omnifig.config.nodes.**ConfigNode**(**args*, *reporter=None*, *settings=None*, *project=None*, *manager=None*, ***kwargs*)

Bases: AutoTreeNode, *AbstractConfig*

The main config node class. This class is used to represent the config tree and is the main interface for interacting with the config.

Settings

alias of OrderedDict

classmethod from_raw(*raw*, ***, *parent=<class 'omnibelt.typing.unspecified_argument'>*, *parent_key=None*, ***kwargs*)

Converts the given raw data into a config node. This will recursively convert all nested data into config nodes.

Parameters

- **raw** (Any) – python data to convert (may be a primitive or a dict/list-like object)
- **parent** (Optional[*ConfigNode*]) – the parent node (if any) of the node to be created
- **parent_key** (Optional[str]) – the key of the node to be created (if any) in the parent node
- ****kwargs** – additional arguments to pass to the constructor

Return type
ConfigNode

Returns
The created config node

class **Search**(*origin*, *queries*, *default*, *parent_search=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Bases: *AbstractSearch*

Used to traverse the config tree and find the node corresponding to the given set of queries.

confidential_prefix = '_'

force_create_prefix = '<!>'

delegation_prefix = '<>'

delegation_origin_prefix = '<o>'

missing_key_payload = '__x__'

sub_search()

Returns a context manager that allows new searches to reference back to this search

Return type
ContextManager

__init__(*origin*, *queries*, *default*, *parent_search=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Evaluates the given queries and default values to find the node (or consequent product) in the config tree.

Parameters

- **origin** (*ConfigNode*) – the node from which to start the search
- **queries** (Optional[Sequence[str]]) – keys to search for (in order) in the config tree
- **default** (Any) – if the search fails, this value is returned
- **parent_search** (Optional[*Search*]) – if the search is nested, this is the search that was used to find the parent node
- ****kwargs** – additional arguments to pass to the constructor (not used)

find_node(*silent=None*)

Traverses the config tree from the origin node to find the node corresponding to the queries (given in __init__). If no queries are provided, the origin node is returned.

This method should be used when the end result of the search should be the node (not a product).

Return type
ConfigNode

Returns
The node corresponding to the queries with the current search as the trace (for reporting)

Raises
SearchFailed – if the node could not be found and no default was provided

find_product(*silent=None*)

Traverses the config tree from the origin node to find the node corresponding to the queries and then produces the product of that node (using the "process" or "create" method of the node).

This method should be used when the end result of the search should be the product (i.e. value contained by the node).

If all the queries failed, the default is returned (if provided) and reported using the origin's reporter.

Parameters
silent (Optional[bool]) – propagate the silent flag the reporter or node processing

Return type
Any

Returns
Result of resolving the queries, which is either the product of the node or the default value if no node was found

Raises
SearchFailed – if the node could not be found and no default was provided

process_node(*node*)

Processes the node by checking for delegations (and then resolving those) or any other special search features (such as making sure this node is still valid).

A delegation is a node where the value is itself interpretted as a query to a different node (e.g. using the prefix "<>"). There are a few specific prefixes that can be used for different types of delegations:

- <>: delegate to a new node starting the search from the current node
- <o>: delegate to a new node starting the search from the origin node
- **<!>: delegate to a new node for which the product is always newly created** (rather than reused if it already exists)

If the current node delegates, the unused queries and query chain may be updated through the `_resolve_query()` method.

If the value of a node is "__x__", the node is considered to be missing (i.e. it effectively doesn't exist).

Parameters
node (*ConfigNode*) – the result of resolving the queries thus far

Return type
ConfigNode

Returns
The node once all delegations and special features have been resolved

Raises
SearchFailed – if the node is not valid or a delegation could not be resolved

exception SearchFailed(**queries*)

Bases: *SearchFailed*

Raised when a search fails to find a value

__init__(**queries*)

class Reporter(*indent=' > ', flair='| ', alias_fmt=' --> ', colon=': ', max_num_aliases=3, **kwargs*)

Bases: *AbstractReporter*

Formats and prints the results of a search over the config tree.

__init__(*indent=' > ', flair='| ', alias_fmt=' --> ', colon=': ', max_num_aliases=3, **kwargs*)

Specifies the format in which the search results are printed to the console.

Parameters
- **indent** (str) – for each depth, this string is prepended to the line
- **flair** (str) – prepended to every line this reporter prints (to distinguish it from other output)
- **alias_fmt** (str) – used to indicate that a query was replaced by another (e.g. due to delegation or missing)
- **colon** (str) – separates the key from the value
- **max_num_aliases** (int) – the maximum number of aliases to print before truncating the list
- ****kwargs** – passed to the parent class (unused)

static log(**msg*, *end='\\n'*, *sep=' '*, *silent=None*)

Prints the message to the console (similar to print).

Parameters
- ***msg** (str) – terms to join and print
- **end** (str) – ending character (default is newline)
- **sep** (str) – character to join terms (default is space)
- **silent** (Optional[bool]) – if True, the message is not printed (but still returned)

Return type
str

Returns
The message that was printed

get_key(*trace*)

Formats the key of the node (taking parents into account).

Parameters
trace (*Search*) – the search context (used to get the node)

Return type
str

Returns
The formatted key (including aliases and parents)

report_node(*node*, ***, *silent=None*)

Reports when a node was found and prints it to the console. By default, no message is printed.

Parameters
- **node** (*ConfigNode*) – result of the search
- **silent** (bool) – if True, the message is not printed (but still returned)

Return type
Optional[str]

Returns
None

report_default(*node*, *default*, *, *silent=None*)

Reports when a default value was used and prints it to the console.

Parameters
- **node** (*ConfigNode*) – result of the search
- **default** (Any) – value that was used instead
- **silent** (bool) – if True, the message is not printed (but still returned)

Return type
Optional[str]

Returns
Message that was printed

report_empty(*node*, *, *silent=None*)

Reports when a node was found, but it was empty and prints it to the console.

Parameters
- **node** (*ConfigNode*) – result of the search
- **silent** (bool) – if True, the message is not printed (but still returned)

Return type
Optional[str]

Returns
Message that was printed

report_iterator(*node*, *product=False*, *, *silent=None*)

Reports when a node was found and returned as an iterator and prints it to the console.

Parameters
- **node** (*ConfigNode*) – result of the search
- **product** (Optional[bool]) – if True, the iterator returns the products of the nodes (defaults to False)
- **silent** (Optional[bool]) – if True, the message is not printed (but still returned)

Return type
Optional[str]

Returns
Message that was printed

reuse_product(*node*, *product*, *, *silent=None*)

Reports when a node was found and its product was reused and prints it to the console.

Parameters
- **node** (*ConfigNode*) – result of the search
- **product** (Any) – the product that was reused
- **silent** (bool) – if True, the message is not printed (but still returned)

Return type
Optional[str]

Returns
Message that was printed

create_primitive(*node*, *value=<class 'omnibelt.typing.unspecified_argument'>*, *, *silent=None*)

Reports when a product was created that was a primitive and prints it to the console.

Parameters
- **node** (*ConfigNode*) – result of the search
- **value** (Union[str, int, float, bool, None]) – of the product
- **silent** (bool) – if True, the message is not printed (but still returned)

Return type
Optional[str]

Returns
Message that was printed

`create_container`(*node*, *, *silent=None*)

Reports when a product was created that was a container and prints it to the console.

Parameters
- **node** (*ConfigNode*) – result of the search
- **silent** (bool) – if True, the message is not printed (but still returned)

Return type
Optional[str]

Returns
Message that was printed

`create_component`(*node*, *, *component_type=None*, *modifiers=None*, *creator_type=None*, *silent=None*)

Reports when a product was created that was a component and prints it to the console.

Parameters
- **node** (*ConfigNode*) – result of the search
- **component_type** (str) – registered name of the component
- **modifiers** (Optional[Sequence[str]]) – registered names of the modifiers
- **creator_type** (str) – registered name of the creator (defaults to None)
- **silent** (bool) – if True, the message is not printed (but still returned)

Return type
Optional[str]

Returns
Message that was printed

`exception CycleError`(*config*)

Bases: RuntimeError

Raised when a cycle is detected in the config tree.

`__init__`(*config*)

`class DefaultCreator`(*config*, *, *component_type=<class 'omnibelt.typing.unspecified_argument'>*, *modifiers=None*, *project=None*, *component_entry=<class 'omnibelt.typing.unspecified_argument'>*, *silent=None*, ***kwargs*)

Bases: *AbstractCreator*

Manages the creation of products of config nodes. Generally, there are three types of products:

- primitives (int, float, str, bool, None)
- containers (list, dict)
- components (objects, must be registered, and may optionally be modified)

The default creator is responsible for creating the products, but you can also create your own creators (e.g. subclassses) and then specify them when registering components (to make those the defaults) or in the config directly. Alternatively, you can force a specific creator to be used by changing the config setting "creator".

Note, that it is generally up to the creator to call the config reporter to report the creation of products.

`classmethod replace`(*creator*, *config*, *, *component_type=None*, *modifiers=None*, *project=<class 'omnibelt.typing.unspecified_argument'>*, *component_entry=<class 'omnibelt.typing.unspecified_argument'>*, *silent=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Extracts information from the given creator to replace it. Used primarily in `DefaultCreator.validate()`.

Return type
AbstractCreator

__init__(*config*, *, *component_type=<class 'omnibelt.typing.unspecified_argument'>*, *modifiers=None*, *project=None*, *component_entry=<class 'omnibelt.typing.unspecified_argument'>*, *silent=None*, ***kwargs*)

A default creator is instantiated each time a product of a config node is created.

Parameters
- **config** (*ConfigNode*) – node for which the product is being created
- **component_type** (`Optional[str]`) – if not specified, the type is extracted from the config node with the key "_type"
- **modifiers** (`Optional[Sequence[str]]`) – if not specified, the modifiers are extracted from the config node with the key "_mod"
- **project** (`Optional[`*AbstractProject*`]`) – the owning project, if not specified, the same project as the config is used
- **component_entry** (`Optional[NamedTuple]`) – if the component entry has already been found, it can be passed here
- **silent** (`Optional[bool]`) – if True, suppresses the reporter from printing messages
- ****kwargs** – additional arguments (unused)

validate(*config*)

Validates the creator. If the creator is invalid, a new one is created and returned based on what is specified in the config or component_entry.

If a different creator is specified, the current one is replaced with `DefaultCreator.replace()`. Otherwise, the creator is returned unchanged.

Parameters
config (*AbstractConfig*) – node for which the product is being created

Return type
AbstractCreator

Returns
Validated creator

create_product(*config*, *args=None*, *kwargs=None*, *, *silent=None*)

Top level method for creating the product from the config node which is called by the config node.

Parameters
- **config** (*ConfigNode*) – node for which the product is being created
- **args** (`Optional[Tuple]`) – manual positional arguments to be passed to the component constructor
- **kwargs** (`Optional[Dict[str, Any]]`) – manual keyword arguments to be passed to the component constructor
- **silent** (`Optional[bool]`) – if True, suppresses the reporter from printing messages

Return type
`Any`

Returns
Product created from the config node

search(**queries*, *default=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Creates a search object that can be used to resolve the given queries to find the corresponding node.

Note that this method is usually not called directly, but rather through the top level methods such as `ConfigNode.pull()` or `ConfigNode.peek()`.

Parameters

- ***queries** (str) – list of queries to be resolved
- **default** (Optional[Any]) – if all the queries fail, this value will be returned
- ****kwargs** – additional keyword arguments to be passed to the search object constructor

Return type
Search

Returns
Search object that can be used to traverse the config tree

peeks(**queries*, *default=<class 'omnibelt.typing.unspecified_argument'>*, *silent=None*)

Searches in the config based on the given queries and returns the resulting node.

If multiple queries are given, the first one that resolves to a node will be returned. If all the queries fail, the default value will be returned (if given), otherwise a Search.SearchFailed will be raised.

Parameters

- ***queries** (str) – list of queries to be resolved
- **default** (Optional[Any]) – value to be returned if all the queries fail
- **silent** (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
ConfigNode

Returns
Config node that corresponds to the first query that resolves to a node

Raises
Search.SearchFailed – if all the queries fail and no default value is given

pulls(**queries*, *default=<class 'omnibelt.typing.unspecified_argument'>*, *silent=None*, ***kwargs*)

Searches in the config based on the given queries and returns the product of the resulting node. The product is the value that node contains, which can be a primitive, a container, or a component. If the product is a container or component, it will be created if it has not been created yet.

If multiple queries are given, the first one that resolves to a node is used If all the queries fail, the default value will be returned (if given), otherwise a Search.SearchFailed will be raised.

Parameters

- ***queries** (str) – list of queries to be resolved
- **default** (Optional[Any]) – value to be returned if all the queries fail
- **silent** (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
Any

Returns
Product of the config node corresponding to the first query that doesn't fail

Raises
Search.SearchFailed – if all the queries fail and no default value is given

push(*addr*, *value*, *overwrite=True*, *silent=None*)

Inserts a new node into the config tree.

Parameters

- **addr** (str) – of the new node (i.e. the key or index)

- **value** (Any) – of the new node
- **overwrite** (bool) – if True, overwrites the existing node if there is one
- **silent** (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
bool

Returns
True if the node was inserted, False if it was not inserted

Raises
ReadonlyError – if the config is readonly

peek_children(*, *silent=None*)

Returns an iterator over the child nodes of self.

The iterator skips invalid children (such as empty values or __x__).

Parameters
silent (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
Iterator[*ConfigNode*]

Returns
Iterator over the children of self

peek_named_children(*, *silent=None*)

Returns an iterator over the child nodes of self, including their keys.

The iterator skips invalid children (such as empty values or __x__).

Parameters
silent (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
Iterator[Tuple[str, *ConfigNode*]]

Returns
Iterator producing tuples in the form of (key, child_node)

pull_children(*, *force_create=False*, *silent=None*)

Returns an iterator over the products of the child nodes of self.

The iterator skips invalid children (such as empty values or __x__).

Parameters

- **force_create** (Optional[bool]) – if True, will always create new products instead of reuse existing ones
- **silent** (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
Iterator[Any]

Returns
Iterator over the products of the children of the node

pull_named_children(*, *force_create=False*, *silent=None*)

Returns an iterator over the products of the child nodes of self, including their keys.

The iterator skips invalid children (such as empty values or __x__).

Parameters

- **force_create** (Optional[bool]) – if True, will always create new products instead of reuse existing ones
- **silent** (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
Iterator[Tuple[str, Any]]

Returns
Iterator over the products of the children of the node

peek_process(*query*, *default=<class 'omnibelt.typing.unspecified_argument'>*, **args*, ***kwargs*)

Convenience method which composes `peek()` and *process()*, to only process a node if it exists, and otherwise return the given default value.

Parameters

- **query** – of the node to be processed
- **default** (Optional[Any]) – value to be returned if the node doesn't exist
- ***args** (Any) – positional arguments to be passed to *process()* (e.g. to the constructor of the component)
- ****kwargs** (Any) – keyword arguments to be passed to *process()* (e.g. to the constructor of the component)

Return type
Any

Returns
Product of the node corresponding to the query, or the default value if the node doesn't exist

Raises
Search.SearchFailed – if the query fails and no default value is given

peeks_process(**queries*, *default=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Convenience method which composes *peeks()* and *process()*, to only process a node if it exists, and otherwise return the given default value.

Note that since this method allows for multiple queries, no positional arguments can be passed to the *process()* method, and neither can the keyword argument `default`.

Parameters

- ***queries** – of the nodes to be processed
- **default** (Optional[Any]) – value to be returned if the node doesn't exist
- ****kwargs** (Any) – keyword arguments to be passed to *process()* (e.g. to the constructor of the component)

Return type
Any

Returns
Product of the node corresponding to the query, or the default value if the node doesn't exist

Raises
Search.SearchFailed – if the query fails and no default value is given

peek_create(*query*, *default=<class 'omnibelt.typing.unspecified_argument'>*, **args*, ***kwargs*)

Convenience method which composes `peek()` and *create()*, to only create the product if the node exists, and otherwise return the given default value.

Parameters

- **query** – of the node for which the product should be created
- **default** (Optional[Any]) – value to be returned if the node doesn't exist
- ***args** (Any) – positional arguments to be passed to *create()* (e.g. to the constructor of the component)
- ****kwargs** (Any) – keyword arguments to be passed to *create()* (e.g. to the constructor of the component)

Return type
Any

Returns
Product of the node corresponding to the query, or the default value if the node doesn't exist

Raises
Search.SearchFailed – if the query fails and no default value is given

peeks_create(**queries*, *default=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Convenience method which composes *peeks()* and *create()*, to only create the product if the node exists, and otherwise return the given default value.

Note that since this method allows for multiple queries, no positional arguments can be passed to the *create()* method, and neither can the keyword argument `default`.

Parameters

- ***queries** – of the nodes to be processed
- **default** (Optional[Any]) – value to be returned if the node doesn't exist
- ****kwargs** (Any) – keyword arguments to be passed to *create()* (e.g. to the constructor of the component)

Return type
Any

Returns
Product of the node corresponding to the query, or the default value if the node doesn't exist

Raises
Search.SearchFailed – if the query fails and no default value is given

__init__(**args*, *reporter=None*, *settings=None*, *project=None*, *manager=None*, ***kwargs*)

Initializes a new config node. Most of the arguments passed to this constructor should usually be `None` so that this node refers to the values of the root config node.

Parameters

- ***args** – unused positional arguments passed to the constructor of the super class
- **reporter** (Optional[*Reporter*]) – used for reporting when data is accessed, defers to the root node
- **settings** (Optional[OrderedDict]) – used to determine the behavior of searching and producing products, defers to the root node

- **project** (Optional[*AbstractProject*]) – associated with this config, defers to the root node
- **manager** (Optional[*AbstractConfigManager*]) – associated with this config, defers to the root node
- ****kwargs** – unused keyword arguments passed to the constructor of the super class

export(*name*, *, *root=None*, *fmt=None*)

Exports the given config to the given path (in yaml format).

Parameters

- **config** – object to export
- **name** (Union[str, Path]) – of file name or path to export to (without extension)
- **root** (Union[str, Path, None]) – directory to export to (if not provided, the current working directory is used)
- **fmt** (Optional[str]) – format to export to (if not provided, the extension of the file name is used, and defaults to yaml)

Return type
Optional[Path]

Returns
The path to which the config was exported

property project

Returns the project associated with this config tree.

property cro: Tuple[str, ...]

Returns the list of all config files that were composed to produce this config tree. Analogous to the method resolution order (mro) for classes.

Return type
Tuple[str, ...]

property bases: Tuple[str, ...]

Returns the list of config files that were explicitly mentioned to produce this config tree. Analogous to __bases__ for classes.

Return type
Tuple[str, ...]

property manager

Returns the manager associated with this config tree.

property trace: *Search* | None

Returns the current search trace associated this config node (generally only used by creators and reporters).

Return type
Optional[*Search*]

property reporter: *Reporter*

Returns the reporter associated with this config tree.

Return type
Reporter

property settings: OrderedDict

Returns the (global) settings associated with this config tree.

Return type
OrderedDict

property silent: bool

Returns whether this config node is in silent mode.

Return type
bool

exception ReadOnlyError

Bases: Exception

Raised when a read-only config node is attempted to be modified.

class ConfigContext(*config*, *settings*)

Bases: object

Context manager for temporarily modifying the (global) settings of a config tree.

__init__(*config*, *settings*)

context(***settings*)

Returns a context manager for temporarily modifying the (global) settings of this config tree.

Return type
ContextManager

create(**args*, ***kwargs*)

Creates a new value based on the contents of self.

Parameters

- ***args** (Any) – Manual arguments to pass to the value constructor.
- ****kwargs** (Any) – Manual keyword arguments to pass to the value constructor.

Return type
Any

Returns
The newly created value.

create_silent(**args*, ***kwargs*)

Convenience method for creating a value in silent mode.

Return type
Any

process(**args*, ***kwargs*)

Processes the config object using the contents of self.

If a value for this config object has already been created, it is returned instead of creating a new one.

Parameters

- ***args** (Any) – Manual arguments to pass to the value constructor. (ignored if a value has already been created)
- ****kwargs** (Any) – Manual keyword arguments to pass to the value constructor.
- **created)** (*(ignored if a value has already been*) –

Return type
Any

Returns
The processed value.

process_silent(**args*, ***kwargs*)

Convenience method for processing a value in silent mode.

Return type
Any

property product_exists: bool

Returns True if the product of this config node has already been created.

Return type
bool

clear_product(*recursive=True*)

Removes the product of this config node (if it exists), and optionally removes the products of all child nodes.

Parameters
recursive (bool) – if True, the products of all child nodes are also removed

Return type
None

to_yaml(*stream=None*, *default_flow_style=None*, *sort_keys=True*, ***kwargs*)

Dumps the contents of this config node in YAML format to the specified stream.

Parameters

- **stream** – destination stream
- **default_flow_style** – YAML formatting option (see PyYAML documentation)
- **sort_keys** – YAML formatting option (if True, keys are sorted alphabetically)
- ****kwargs** (Any) – additional keyword arguments to pass to PyYAML's dump function

Return type
None

Returns
None

update(*update*, ***, *clear_product=True*)

Updates the contents of this config node with the contents of another config node.

Parameters

- **update** (*ConfigNode*) – the config node to update from
- **clear_product** (bool) – if True, all references to the products of self and update are removed

Return type
ConfigNode

Returns
The updated config node, self

validate()

Recursively validates the contents of this config node.

Including removing any children with the value `_x_` (marked for removal).

silence(*silent=True*)

Convenience method for temporarily setting the silent flag of this config node.

Return type
ContextManager

print(**terms*, *force=False*, *sep=' '*, *end='\\n'*, ***kwargs*)

Convenience method for printing iff (*force* or not self.silent)

DefaultNode

alias of *ConfigSparseNode*

DenseNode

alias of *ConfigDenseNode*

SparseNode

alias of *ConfigSparseNode*

class omnifig.config.nodes.**ConfigSparseNode**(**args*, *reporter=None*, *settings=None*, *project=None*, *manager=None*, ***kwargs*)

Bases: AutoTreeSparseNode, *ConfigNode*

A config node that treats its children as being in a dict.

class omnifig.config.nodes.**ConfigDenseNode**(**args*, *reporter=None*, *settings=None*, *project=None*, *manager=None*, ***kwargs*)

Bases: AutoTreeDenseNode, *ConfigNode*

A config node that treats its children as being in a list.

1.21 Configurable

class omnifig.configurable.**Configurable**

Bases: *AbstractConfigurable*, Modifiable

Mix-in class for objects that can be constructed with a config object.

It is strongly recommended that components and modifiers inherit from this class to seamlessly fill in missing arguments with the config object.

classmethod init_from_config(*config*, *args=None*, *kwargs=None*, ***, *silent=None*)

Constructor to initialize a class informed by the config object *config*. This will run the usual constructor `__init__`, except any arguments that are missing from the signature will be filled in from the config.

Parameters

- **config** (*AbstractConfig*) – Config object to use
- **args** (Optional[Tuple]) – Manually specified arguments
- **kwargs** (Optional[Dict[str, Any]]) – Manually specified keyword arguments
- **silent** (Optional[bool]) – If True, no messages are reported when querying the config object.

Return type
Any

Returns
The initialized object

class omnifig.configurable.**Certifiable**

Bases: *Configurable*, *AbstractCertifiable*

Simple mix-in to make the initialization of classes through the config a two-stage process. The first stage calls the `__init__` method, and then after that is complete, the `__certify__` method is called, which can return a new object to replace the original one.

Note, that `__certify__` is only called if the object is initialized through the config (e.g. through `pull()`).

class omnifig.configurable.**silent_config_args**(**args*)

Bases: `object`

Decorator to silence the config when extracting arguments from the config

__init__(**args*)

class omnifig.configurable.**config_aliases**(***aliases*)

Bases: `object`

Method decorator to add aliases to the config arguments of a method

__init__(***aliases*)

Config aliases to store for the method

Parameters

- ****aliases** (`Union[Sequence[str], str]`) – Mapping of aliases: keys should be the name of the argument in the method signature,
- **object.** (*and the values should be the name of the argument to query in the config*) –

1.22 Default Projects

class omnifig.organization.default.**Project**(*path=None*, ***, *creator_registry=None*, *component_registry=None*, *modifier_registry=None*, ***kwargs*)

Bases: *GeneralProject*

class Creator_Registry(**args*, *_sister_registry_object=None*, *_sister_registry_cls=None*, ***kwargs*)

Bases: `Class_Registry`

Registry for creators which determine how components are instantiated from the config.

entry_cls

alias of `Creator_Registry_Entry`

class Component_Registry(**args*, *_sister_registry_object=None*, *_sister_registry_cls=None*, ***kwargs*)

Bases: `Class_Registry`

Registry for components (classes) which can be instantiated through the config.

entry_cls

alias of Component_Registry_Entry

class Modifier_Registry(**args*, *_sister_registry_object=None*, *_sister_registry_cls=None*, ***kwargs*)

Bases: Class_Registry

Registry for modifiers (classes) which can modify components through the config by dynamically defining subclasses.

entry_cls

alias of Modifier_Registry_Entry

__init__(*path=None*, ***, *creator_registry=None*, *component_registry=None*, *modifier_registry=None*, ***kwargs*)

Loads the info if the provided *data* is a file path.

Parameters

- **data** – A file path to a yaml file, or a dictionary containing the info
- ****kwargs** – Other arguments passed on to super()

related()

Iterate over all projects related to this one (based on related in the project info file).

Return type
Iterator[*AbstractProject*]

Returns
An iterator over all projects related to this one.

nonlocal_projects()

Iterator over all projects that are related to this one, followed by all active base projects of the profile (without repeating any projects).

Return type
Iterator[NamedTuple]

Returns
An iterator over all projects that are related to this one, followed by all active base projects

xray(*artifact*, ***, *include_nonlocal=True*, *as_list=False*, *sort=False*, *reverse=False*)

Prints a list of all artifacts of the given type accessible from this project (including related and active base projects).

Parameters

- **artifact** (str) – artifact type (e.g. 'script', 'config')
- **include_nonlocal** (Optional[bool]) – whether to include artifacts from non-local projects (related and active base projects)
- **as_list** (Optional[bool]) – instead of printing, return the list of artifacts
- **sort** (Optional[bool]) – sort the list of artifacts by name
- **reverse** (Optional[bool]) – reverse the order of the list of artifacts

Returns
if as_list is True, returns a list of artifacts

Return type
list

Raises
UnknownArtifactTypeError – if the given artifact type does not exist

`missing_related()`

Iterate over all projects related to this one that cannot be found by the current profile.

Return type
`Iterator[str]`

Returns
An iterator over all projects related to this one that cannot be found by the current profile.

`find_creator(name, default=<class 'omnibelt.typing.unspecified_argument'>)`

Finds the creator with the given name.

Parameters

- **name** (`str`) – the creator was registered with.
- **default** (`Optional[Any]`) – default value to return if the creator is not found.

Return type
`NamedTuple`

Returns
The creator entry corresponding to the given name.

Raises
UnknownArtifactError – If the creator is not found and no default is given.

`register_creator(name, typ, *, description=None)`

Register a creator with the given name.

Parameters

- **name** (`str`) – to register the script under
- **typ** (`Type[AbstractCreator]`) – the creator type (should be a subclass of `AbstractCreator`)
- **description** (`Optional[str]`) – description of the creator

Return type
`NamedTuple`

Returns
The entry of the creator that was registered

`iterate_creators()`

Iterates over all registered creator entries.

Return type
`Iterator[NamedTuple]`

Returns
An iterator over all registered creator entries.

`find_component(name, default=<class 'omnibelt.typing.unspecified_argument'>)`

Finds the component with the given name.

Parameters

- **name** (`str`) – the component was registered with.
- **default** (`Optional[Any]`) – default value to return if the component is not found.

Return type
NamedTuple

Returns
The component entry corresponding to the given name.

Raises
UnknownArtifactError – If the component is not found and no default is given.

register_component(*name*, *typ*, *, *creator=None*, *description=None*)

Register a component with the given name.

Parameters

- **name** (str) – to register the component under
- **typ** (Type) – the component type (recommended to be a subclass of Configurable)
- **creator** (Union[str, *AbstractCreator*]) – the creator to use for this component (if none is specified in the config)
- **description** (Optional[str]) – description of the component

Return type
NamedTuple

Returns
The entry of the component that was registered

iterate_components()

Iterates over all registered component entries.

Return type
Iterator[NamedTuple]

Returns
An iterator over all registered component entries.

find_modifier(*name*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the modifier with the given name.

Parameters

- **name** (str) – the modifier was registered with.
- **default** (Optional[Any]) – default value to return if the modifier is not found.

Return type
NamedTuple

Returns
The modifier entry corresponding to the given name.

Raises
UnknownArtifactError – If the modifier is not found and no default is given.

register_modifier(*name*, *typ*, *, *description=None*)

Register a modifier with the given name.

Parameters

- **name** (str) – to register the modifier under
- **typ** (Type) – the modifier type (recommended to be a subclass of Configurable)

- **description** (Optional[str]) – description of the modifier

Return type
NamedTuple

Returns
The entry of the modifier that was registered

iterate_modifiers()

Iterates over all registered modifiers entries.

Return type
Iterator[NamedTuple]

Returns
An iterator over all registered modifier entries.

find_artifact(*artifact_type*, *ident*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the artifact of the given type and registered with the given identifier by searching first in the current project, then any related projects, and finally any active base projects specified by the profile. Note, that if the artifact ident may specify a project to search in by prefixing the ident with the project name and a colon (e.g. proj:ident).

Parameters

- **artifact_type** (str) – The type of artifact to find.
- **ident** (str) – The identifier of the artifact to find.
- **default** (Optional[Any]) – The default value to return if the artifact is not found.

Return type
NamedTuple

Returns
The artifact entry from the registry corresponding to the given type.

Raises

- *UnknownArtifactTypeError* – If the artifact type is not registered.
- *UnknownArtifactError* – If the artifact is not found and no default is given.
- *UnknownProjectError* – If the project specified in the artifact ident is not found.

class omnifig.organization.default.**Profile**(*data=None*)

Bases: *ProfileBase*

class Project(*path=None*, *, *creator_registry=None*, *component_registry=None*, *modifier_registry=None*, ***kwargs*)

Bases: *GeneralProject*

class Component_Registry(**args*, *_sister_registry_object=None*, *_sister_registry_cls=None*, ***kwargs*)

Bases: Class_Registry

Registry for components (classes) which can be instantiated through the config.

entry_cls

alias of Component_Registry_Entry

class Creator_Registry(*args, _sister_registry_object=None, _sister_registry_cls=None, **kwargs)

Bases: Class_Registry

Registry for creators which determine how components are instantiated from the config.

entry_cls

alias of Creator_Registry_Entry

class Modifier_Registry(*args, _sister_registry_object=None, _sister_registry_cls=None, **kwargs)

Bases: Class_Registry

Registry for modifiers (classes) which can modify components through the config by dynamically defining subclasses.

entry_cls

alias of Modifier_Registry_Entry

__init__(*path=None*, *, *creator_registry=None*, *component_registry=None*, *modifier_registry=None*, **kwargs)

Loads the info if the provided *data* is a file path.

Parameters

- **data** – A file path to a yaml file, or a dictionary containing the info
- ****kwargs** – Other arguments passed on to super()

find_artifact(*artifact_type*, *ident*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the artifact of the given type and registered with the given identifier by searching first in the current project, then any related projects, and finally any active base projects specified by the profile. Note, that if the artifact ident may specify a project to search in by prefixing the ident with the project name and a colon (e.g. proj:ident).

Parameters

- **artifact_type** (str) – The type of artifact to find.
- **ident** (str) – The identifier of the artifact to find.
- **default** (Optional[Any]) – The default value to return if the artifact is not found.

Return type

NamedTuple

Returns

The artifact entry from the registry corresponding to the given type.

Raises

- *UnknownArtifactTypeError* – If the artifact type is not registered.
- *UnknownArtifactError* – If the artifact is not found and no default is given.
- *UnknownProjectError* – If the project specified in the artifact ident is not found.

find_component(*name*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the component with the given name.

Parameters

- **name** (str) – the component was registered with.
- **default** (Optional[Any]) – default value to return if the component is not found.

Return type

NamedTuple

Returns

The component entry corresponding to the given name.

Raises

UnknownArtifactError – If the component is not found and no default is given.

`find_creator`(*name*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the creator with the given name.

Parameters
- **name** (`str`) – the creator was registered with.
- **default** (`Optional[Any]`) – default value to return if the creator is not found.

Return type
`NamedTuple`

Returns
The creator entry corresponding to the given name.

Raises
UnknownArtifactError – If the creator is not found and no default is given.

`find_modifier`(*name*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the modifier with the given name.

Parameters
- **name** (`str`) – the modifier was registered with.
- **default** (`Optional[Any]`) – default value to return if the modifier is not found.

Return type
`NamedTuple`

Returns
The modifier entry corresponding to the given name.

Raises
UnknownArtifactError – If the modifier is not found and no default is given.

`iterate_components`()

Iterates over all registered component entries.

Return type
`Iterator[NamedTuple]`

Returns
An iterator over all registered component entries.

`iterate_creators`()

Iterates over all registered creator entries.

Return type
`Iterator[NamedTuple]`

Returns
An iterator over all registered creator entries.

`iterate_modifiers`()

Iterates over all registered modifiers entries.

Return type
`Iterator[NamedTuple]`

Returns
An iterator over all registered modifier entries.

`missing_related`()

Iterate over all projects related to this one that cannot be found by the current profile.

Return type
`Iterator[str]`

Returns
An iterator over all projects related to this one that cannot be found by the current profile.

`nonlocal_projects`()

Iterator over all projects that are related to this one, followed by all active base projects of the profile (without repeating any projects).

Return type
Iterator[NamedTuple]
Returns
An iterator over all projects that are related to this one, followed by all active base projects

register_component(*name*, *typ*, *, *creator=None*, *description=None*)

Register a component with the given name.
Parameters
- **name** (str) – to register the component under
- **typ** (Type) – the component type (recommended to be a subclass of Configurable)
- **creator** (Union[str, *AbstractCreator*]) – the creator to use for this component (if none is specified in the config)
- **description** (Optional[str]) – description of the component

Return type
NamedTuple
Returns
The entry of the component that was registered

register_creator(*name*, *typ*, *, *description=None*)

Register a creator with the given name.
Parameters
- **name** (str) – to register the script under
- **typ** (Type[*AbstractCreator*]) – the creator type (should be a subclass of AbstractCreator)
- **description** (Optional[str]) – description of the creator

Return type
NamedTuple
Returns
The entry of the creator that was registered

register_modifier(*name*, *typ*, *, *description=None*)

Register a modifier with the given name.
Parameters
- **name** (str) – to register the modifier under
- **typ** (Type) – the modifier type (recommended to be a subclass of Configurable)
- **description** (Optional[str]) – description of the modifier

Return type
NamedTuple
Returns
The entry of the modifier that was registered

related()

Iterate over all projects related to this one (based on related in the project info file).
Return type
Iterator[*AbstractProject*]
Returns
An iterator over all projects related to this one.

xray(*artifact*, *, *include_nonlocal=True*, *as_list=False*, *sort=False*, *reverse=False*)

Prints a list of all artifacts of the given type accessible from this project (including related and active base projects).
Parameters
- **artifact** (str) – artifact type (e.g. 'script', 'config')
- **include_nonlocal** (Optional[bool]) – whether to include artifacts from non-local projects (related and active base projects)

- **as_list** (Optional[bool]) – instead of printing, return the list of artifacts
- **sort** (Optional[bool]) – sort the list of artifacts by name
- **reverse** (Optional[bool]) – reverse the order of the list of artifacts

Returns
if as_list is True, returns a list of artifacts

Return type
list

Raises
UnknownArtifactTypeError – if the given artifact type does not exist

__init__(*data=None*)

Loads the info if the provided *data* is a file path.

Parameters

- **data** (Union[str, Path, Dict, None]) – A file path to a yaml file, or a dictionary containing the info
- ****kwargs** – Other arguments passed on to super()

property projects: List[*Project*]

Convenience property for accessing the projects in the profile. Recommended for debugging only.

Return type
List[*Project*]

initialize(**projects*, ***kwargs*)

Initializes the profile by activating it and then activating all projects specified, also adds the projects to the profile's active base projects.

Parameters

- ***projects** (str) – The names of projects to activate and add to the active base projects.
- ****kwargs** (Any) – Additional keyword arguments to pass to the project initialization methods.

Return type
None

Returns
None

iterate_base_projects()

Iterates through the active base projects in the profile.

The active base projects are those specified in the profile's active-projects list, and are expected to be fallback projects for the current project for finding artifacts.

Return type
Iterator[*Project*]

Returns
An iterator over the active base projects.

iterate_projects()

Iterates through the projects in the profile in the order they were created. Note that this iterator will not include duplicates, even if the same project is loaded under multiple names.

Return type
Iterator[*Project*]

Returns
An iterator over the projects in the profile.

exception UnknownProjectError

Bases: KeyError

Raised when trying to get a project with an invalid path.

get_project(*ident=None*, *is_current=None*)

Gets the project with the given name/path. If no name/path is given, then the current project is returned. If the specified project has not been initialized, then it is created using the profile's Project class (but it is not activated).

If a name is given, it must be the name of a project that has already been loaded, or it must be a key in the dict *projects* the profile's info file where the value is the corresponding path.

If a path is given, the path may either be a yaml file (interpretted as the project's info file) or a directory (in which case the project's info file is assumed to be named *.fig.project.yaml*).

Parameters

- **ident** (Union[str, Path]) – of the project to get. If not given, then the current project is returned.
- **is_current** (Optional[bool]) – if True, then the project is set as the current project.

Return type
AbstractProject

Returns
The project with the given name/path.

Raises

- ***UnknownProjectError*** – If the project is not found.
- **ValueError** – If the project is found, but it has a name that a different project is already using.

1.23 Exporting Config

class omnifig.exporting.**ConfigExporter**

Bases: SimpleExporterBase

Exporter for config objects, can load and save config objects in four formats: json, yaml, toml, and the native ".fig.yml" format (which is equivalent to yaml).

export_payload(*src*, *payload*, *path*, ***, *fmt=None*, ***kwargs*)

Exports the given payload to the given path.

Parameters

- **src** (AbstractExportManager) – manager used for delegating the export to a different format
- **payload** (*ConfigNode*) – config object to be exported
- **path** (Path) – destination path for the export
- **fmt** (Optional[str]) – format to use for the export (if None, will be inferred from the path, defaults to *.fig.yml*)

Return type
Path

Returns
the path to the exported file (or None if the export failed)

1.24 Config Manager

class omnifig.config.manager.**ConfigManager**(*project*)

Bases: *AbstractConfigManager*

class **ConfigNode**(**args*, *reporter=None*, *settings=None*, *project=None*, *manager=None*, ***kwargs*)

Bases: AutoTreeNode, *AbstractConfig*

The main config node class. This class is used to represent the config tree and is the main interface for interacting with the config.

class **ConfigContext**(*config*, *settings*)

Bases: object

Context manager for temporarily modifying the (global) settings of a config tree.

__init__(*config*, *settings*)

exception **CycleError**(*config*)

Bases: RuntimeError

Raised when a cycle is detected in the config tree.

__init__(*config*)

class **DefaultCreator**(*config*, ***, *component_type=<class 'omnibelt.typing.unspecified_argument'>*, *modifiers=None*, *project=None*, *component_entry=<class 'omnibelt.typing.unspecified_argument'>*, *silent=None*, ***kwargs*)

Bases: *AbstractCreator*

Manages the creation of products of config nodes. Generally, there are three types of products:

- primitives (int, float, str, bool, None)
- containers (list, dict)
- components (objects, must be registered, and may optionally be modified)

The default creator is responsible for creating the products, but you can also create your own creators (e.g. subclassses) and then specify them when registering components (to make those the defaults) or in the config directly. Alternatively, you can force a specific creator to be used by changing the config setting "creator".

Note, that it is generally up to the creator to call the config reporter to report the creation of products.

__init__(*config*, ***, *component_type=<class 'omnibelt.typing.unspecified_argument'>*, *modifiers=None*, *project=None*, *component_entry=<class 'omnibelt.typing.unspecified_argument'>*, *silent=None*, ***kwargs*)

A default creator is instantiated each time a product of a config node is created.

Parameters

- **config** (*ConfigNode*) – node for which the product is being created
- **component_type** (Optional[str]) – if not specified, the type is extracted from the config node with the key "_type"
- **modifiers** (Optional[Sequence[str]]) – if not specified, the modifiers are extracted from the config node with the key "_mod"

- **project** (Optional[*AbstractProject*]) – the owning project, if not specified, the same project as the config is used
- **component_entry** (Optional[NamedTuple]) – if the component entry has already been found, it can be passed here
- **silent** (Optional[bool]) – if True, suppresses the reporter from printing messages
- ****kwargs** – additional arguments (unused)

create_product(*config*, *args=None*, *kwargs=None*, ***, *silent=None*)

Top level method for creating the product from the config node which is called by the config node.

Parameters

- **config** (*ConfigNode*) – node for which the product is being created
- **args** (Optional[Tuple]) – manual positional arguments to be passed to the component constructor
- **kwargs** (Optional[Dict[str, Any]]) – manual keyword arguments to be passed to the component constructor
- **silent** (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type

Any

Returns

Product created from the config node

classmethod replace(*creator*, *config*, ***, *component_type=None*, *modifiers=None*, *project=<class 'omnibelt.typing.unspecified_argument'>*, *component_entry=<class 'omnibelt.typing.unspecified_argument'>*, *silent=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Extracts information from the given creator to replace it. Used primarily in `DefaultCreator.validate()`.

Return type

AbstractCreator

validate(*config*)

Validates the creator. If the creator is invalid, a new one is created and returned based on what is specified in the config or component_entry.

If a different creator is specified, the current one is replaced with `DefaultCreator.replace()`. Otherwise, the creator is returned unchanged.

Parameters

config (*AbstractConfig*) – node for which the product is being created

Return type

AbstractCreator

Returns

Validated creator

DefaultNode

alias of *ConfigSparseNode*

DenseNode

alias of *ConfigDenseNode*

exception ReadOnlyError

Bases: Exception

Raised when a read-only config node is attempted to be modified.

class Reporter(*indent=' > '*, *flair='| '*, *alias_fmt=' --> '*, *colon=': '*, *max_num_aliases=3*, ***kwargs*)

Bases: *AbstractReporter*

Formats and prints the results of a search over the config tree.

`__init__`(*indent=' > '*, *flair='| '*, *alias_fmt=' --> '*, *colon=': '*, *max_num_aliases=3*, ***kwargs*)

Specifies the format in which the search results are printed to the console.

Parameters

- **indent** (`str`) – for each depth, this string is prepended to the line
- **flair** (`str`) – prepended to every line this reporter prints (to distinguish it from other output)
- **alias_fmt** (`str`) – used to indicate that a query was replaced by another (e.g. due to delegation or missing)
- **colon** (`str`) – separates the key from the value
- **max_num_aliases** (`int`) – the maximum number of aliases to print before truncating the list
- ****kwargs** – passed to the parent class (unused)

`create_component`(*node*, ***, *component_type=None*, *modifiers=None*, *creator_type=None*, *silent=None*)

Reports when a product was created that was a component and prints it to the console.

Parameters

- **node** (*ConfigNode*) – result of the search
- **component_type** (`str`) – registered name of the component
- **modifiers** (`Optional[Sequence[str]]`) – registered names of the modifiers
- **creator_type** (`str`) – registered name of the creator (defaults to None)
- **silent** (`bool`) – if True, the message is not printed (but still returned)

Return type

`Optional[str]`

Returns

Message that was printed

`create_container`(*node*, ***, *silent=None*)

Reports when a product was created that was a container and prints it to the console.

Parameters

- **node** (*ConfigNode*) – result of the search
- **silent** (`bool`) – if True, the message is not printed (but still returned)

Return type

`Optional[str]`

Returns

Message that was printed

`create_primitive`(*node*, *value=<class 'omnibelt.typing.unspecified_argument'>*, ***, *silent=None*)

Reports when a product was created that was a primitive and prints it to the console.

Parameters

- **node** (*ConfigNode*) – result of the search
- **value** (`Union[str, int, float, bool, None]`) – of the product
- **silent** (`bool`) – if True, the message is not printed (but still returned)

Return type

`Optional[str]`

Returns

Message that was printed

`get_key`(*trace*)

Formats the key of the node (taking parents into account).

Parameters

trace (*Search*) – the search context (used to get the node)

Return type
str
Returns
The formatted key (including aliases and parents)

static log(**msg*, *end='\\n'*, *sep=' '*, *silent=None*)

Prints the message to the console (similar to `print`).

Parameters
- ***msg** (str) – terms to join and print
- **end** (str) – ending character (default is newline)
- **sep** (str) – character to join terms (default is space)
- **silent** (Optional[bool]) – if True, the message is not printed (but still returned)

Return type
str
Returns
The message that was printed

report_default(*node*, *default*, *, *silent=None*)

Reports when a default value was used and prints it to the console.

Parameters
- **node** (*ConfigNode*) – result of the search
- **default** (Any) – value that was used instead
- **silent** (bool) – if True, the message is not printed (but still returned)

Return type
Optional[str]
Returns
Message that was printed

report_empty(*node*, *, *silent=None*)

Reports when a node was found, but it was empty and prints it to the console.

Parameters
- **node** (*ConfigNode*) – result of the search
- **silent** (bool) – if True, the message is not printed (but still returned)

Return type
Optional[str]
Returns
Message that was printed

report_iterator(*node*, *product=False*, *, *silent=None*)

Reports when a node was found and returned as an iterator and prints it to the console.

Parameters
- **node** (*ConfigNode*) – result of the search
- **product** (Optional[bool]) – if True, the iterator returns the products of the nodes (defaults to False)
- **silent** (Optional[bool]) – if True, the message is not printed (but still returned)

Return type
Optional[str]
Returns
Message that was printed

report_node(*node*, *, *silent=None*)

Reports when a node was found and prints it to the console. By default, no message is printed.

Parameters
- **node** (*ConfigNode*) – result of the search
- **silent** (bool) – if True, the message is not printed (but still returned)

Return type
Optional[str]
Returns
None

reuse_product(*node*, *product*, *, *silent=None*)

Reports when a node was found and its product was reused and prints it to the console.
Parameters
- **node** (*ConfigNode*) – result of the search
- **product** (Any) – the product that was reused
- **silent** (bool) – if True, the message is not printed (but still returned)

Return type
Optional[str]
Returns
Message that was printed

class Search(*origin*, *queries*, *default*, *parent_search=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Bases: *AbstractSearch*

Used to traverse the config tree and find the node corresponding to the given set of queries.

__init__(*origin*, *queries*, *default*, *parent_search=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Evaluates the given queries and default values to find the node (or consequent product) in the config tree.
Parameters
- **origin** (*ConfigNode*) – the node from which to start the search
- **queries** (Optional[Sequence[str]]) – keys to search for (in order) in the config tree
- **default** (Any) – if the search fails, this value is returned
- **parent_search** (Optional[*Search*]) – if the search is nested, this is the search that was used to find the parent node
- ****kwargs** – additional arguments to pass to the constructor (not used)

confidential_prefix = '_'

delegation_origin_prefix = '<o>'

delegation_prefix = '<>'

find_node(*silent=None*)

Traverses the config tree from the origin node to find the node corresponding to the queries (given in __init__). If no queries are provided, the origin node is returned.

This method should be used when the end result of the search should be the node (not a product).
Return type
ConfigNode
Returns
The node corresponding to the queries with the current search as the trace (for reporting)
Raises
SearchFailed – if the node could not be found and no default was provided

find_product(*silent=None*)

Traverses the config tree from the origin node to find the node corresponding to the queries and then produces the product of that node (using the "process" or "create" method of the node).

This method should be used when the end result of the search should be the product (i.e. value contained by the node).

If all the queries failed, the default is returned (if provided) and reported using the origin's reporter.

Parameters
silent (Optional[bool]) – propagate the silent flag the reporter or node processing

Return type
Any

Returns
Result of resolving the queries, which is either the product of the node or the default value if no node was found

Raises
SearchFailed – if the node could not be found and no default was provided

force_create_prefix = '<!>'

missing_key_payload = '__x__'

process_node(*node*)

Processes the node by checking for delegations (and then resolving those) or any other special search features (such as making sure this node is still valid).

A delegation is a node where the value is itself interpretted as a query to a different node (e.g. using the prefix "<>"). There are a few specific prefixes that can be used for different types of delegations:

- <>: delegate to a new node starting the search from the current node
- <o>: delegate to a new node starting the search from the origin node
- **<!>: delegate to a new node for which the product is always newly created** (rather than reused if it already exists)

If the current node delegates, the unused queries and query chain may be updated through the `_resolve_query()` method.

If the value of a node is "__x__", the node is considered to be missing (i.e. it effectively doesn't exist).

Parameters
node (*ConfigNode*) – the result of resolving the queries thus far

Return type
ConfigNode

Returns
The node once all delegations and special features have been resolved

Raises
SearchFailed – if the node is not valid or a delegation could not be resolved

sub_search()

Returns a context manager that allows new searches to reference back to this search

Return type
ContextManager

exception SearchFailed(**queries*)

Bases: *SearchFailed*

Raised when a search fails to find a value

__init__(**queries*)

Settings

alias of OrderedDict

SparseNode

alias of *ConfigSparseNode*

__init__(**args*, *reporter=None*, *settings=None*, *project=None*, *manager=None*, ***kwargs*)

Initializes a new config node. Most of the arguments passed to this constructor should usually be `None` so that this node refers to the values of the root config node.

Parameters

- ***args** – unused positional arguments passed to the constructor of the super class
- **reporter** (`Optional[Reporter]`) – used for reporting when data is accessed, defers to the root node
- **settings** (`Optional[OrderedDict]`) – used to determine the behavior of searching and producing products, defers to the root node
- **project** (`Optional[AbstractProject]`) – associated with this config, defers to the root node
- **manager** (`Optional[AbstractConfigManager]`) – associated with this config, defers to the root node
- ****kwargs** – unused keyword arguments passed to the constructor of the super class

property bases: Tuple[str, ...]

Returns the list of config files that were explicitly mentioned to produce this config tree. Analogous to `__bases__` for classes.

Return type
`Tuple[str, ...]`

clear_product(*recursive=True*)

Removes the product of this config node (if it exists), and optionally removes the products of all child nodes.

Parameters
recursive (`bool`) – if True, the products of all child nodes are also removed

Return type
`None`

context(***settings*)

Returns a context manager for temporarily modifying the (global) settings of this config tree.

Return type
`ContextManager`

create(**args*, ***kwargs*)

Creates a new value based on the contents of self.

Parameters

- ***args** (`Any`) – Manual arguments to pass to the value constructor.
- ****kwargs** (`Any`) – Manual keyword arguments to pass to the value constructor.

Return type
`Any`

Returns
The newly created value.

`create_silent`(**args*, ***kwargs*)

Convenience method for creating a value in silent mode.

Return type
`Any`

`property cro: Tuple[str, ...]`

Returns the list of all config files that were composed to produce this config tree. Analogous to the method resolution order (`mro`) for classes.

Return type
`Tuple[str, ...]`

`export`(*name*, *, *root=None*, *fmt=None*)

Exports the given config to the given path (in yaml format).

Parameters

- **config** – object to export
- **name** (`Union[str, Path]`) – of file name or path to export to (without extension)
- **root** (`Union[str, Path, None]`) – directory to export to (if not provided, the current working directory is used)
- **fmt** (`Optional[str]`) – format to export to (if not provided, the extension of the file name is used, and defaults to yaml)

Return type
`Optional[Path]`

Returns
The path to which the config was exported

`classmethod from_raw`(*raw*, *, *parent=<class 'omnibelt.typing.unspecified_argument'>*, *parent_key=None*, ***kwargs*)

Converts the given raw data into a config node. This will recursively convert all nested data into config nodes.

Parameters

- **raw** (`Any`) – python data to convert (may be a primitive or a dict/list-like object)
- **parent** (`Optional[ConfigNode]`) – the parent node (if any) of the node to be created
- **parent_key** (`Optional[str]`) – the key of the node to be created (if any) in the parent node
- ****kwargs** – additional arguments to pass to the constructor

Return type
ConfigNode

Returns
The created config node

`property manager`

Returns the manager associated with this config tree.

`peek_children`(*, *silent=None*)

Returns an iterator over the child nodes of `self`.

The iterator skips invalid children (such as empty values or `__x__`).

Parameters
silent (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
Iterator[*ConfigNode*]

Returns
Iterator over the children of self

peek_create(*query*, *default=<class 'omnibelt.typing.unspecified_argument'>*, **args*, ***kwargs*)

Convenience method which composes peek() and *create()*, to only create the product if the node exists, and otherwise return the given default value.

Parameters

- **query** – of the node for which the product should be created
- **default** (Optional[Any]) – value to be returned if the node doesn't exist
- ***args** (Any) – positional arguments to be passed to *create()* (e.g. to the constructor of the component)
- ****kwargs** (Any) – keyword arguments to be passed to *create()* (e.g. to the constructor of the component)

Return type
Any

Returns
Product of the node corresponding to the query, or the default value if the node doesn't exist

Raises
Search.SearchFailed – if the query fails and no default value is given

peek_named_children(*, *silent=None*)

Returns an iterator over the child nodes of self, including their keys.

The iterator skips invalid children (such as empty values or __x__).

Parameters
silent (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
Iterator[Tuple[str, *ConfigNode*]]

Returns
Iterator producing tuples in the form of (key, child_node)

peek_process(*query*, *default=<class 'omnibelt.typing.unspecified_argument'>*, **args*, ***kwargs*)

Convenience method which composes peek() and *process()*, to only process a node if it exists, and otherwise return the given default value.

Parameters

- **query** – of the node to be processed
- **default** (Optional[Any]) – value to be returned if the node doesn't exist
- ***args** (Any) – positional arguments to be passed to *process()* (e.g. to the constructor of the component)
- ****kwargs** (Any) – keyword arguments to be passed to *process()* (e.g. to the constructor of the component)

Return type
Any

Returns
Product of the node corresponding to the query, or the default value if the node doesn't exist

Raises
Search.SearchFailed – if the query fails and no default value is given

peeks(**queries*, *default=<class 'omnibelt.typing.unspecified_argument'>*, *silent=None*)

Searches in the config based on the given queries and returns the resulting node.

If multiple queries are given, the first one that resolves to a node will be returned. If all the queries fail, the default value will be returned (if given), otherwise a `Search.SearchFailed` will be raised.

Parameters

- ***queries** (str) – list of queries to be resolved
- **default** (Optional[Any]) – value to be returned if all the queries fail
- **silent** (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
ConfigNode

Returns
Config node that corresponds to the first query that resolves to a node

Raises
Search.SearchFailed – if all the queries fail and no default value is given

peeks_create(**queries*, *default=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Convenience method which composes *peeks()* and *create()*, to only create the product if the node exists, and otherwise return the given default value.

Note that since this method allows for multiple queries, no positional arguments can be passed to the *create()* method, and neither can the keyword argument `default`.

Parameters

- ***queries** – of the nodes to be processed
- **default** (Optional[Any]) – value to be returned if the node doesn't exist
- ****kwargs** (Any) – keyword arguments to be passed to *create()* (e.g. to the constructor of the component)

Return type
Any

Returns
Product of the node corresponding to the query, or the default value if the node doesn't exist

Raises
Search.SearchFailed – if the query fails and no default value is given

peeks_process(**queries*, *default=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Convenience method which composes *peeks()* and *process()*, to only process a node if it exists, and otherwise return the given default value.

Note that since this method allows for multiple queries, no positional arguments can be passed to the *process()* method, and neither can the keyword argument `default`.

Parameters

- ***queries** – of the nodes to be processed
- **default** (Optional[Any]) – value to be returned if the node doesn't exist
- ****kwargs** (Any) – keyword arguments to be passed to *process()* (e.g. to the constructor of the component)

Return type
Any

Returns
Product of the node corresponding to the query, or the default value if the node doesn't exist

Raises
Search.SearchFailed – if the query fails and no default value is given

print(**terms*, *force=False*, *sep=' '*, *end='\\n'*, ***kwargs*)

Convenience method for printing iff (*force* or not self.silent)

process(**args*, ***kwargs*)

Processes the config object using the contents of self.

If a value for this config object has already been created, it is returned instead of creating a new one.

Parameters

- ***args** (Any) – Manual arguments to pass to the value constructor. (ignored if a value has already been created)
- ****kwargs** (Any) – Manual keyword arguments to pass to the value constructor.
- **created)** (*(ignored if a value has already been*) –

Return type
Any

Returns
The processed value.

process_silent(**args*, ***kwargs*)

Convenience method for processing a value in silent mode.

Return type
Any

property product_exists: bool

Returns True if the product of this config node has already been created.

Return type
bool

property project

Returns the project associated with this config tree.

pull_children(*, *force_create=False*, *silent=None*)

Returns an iterator over the products of the child nodes of `self`.

The iterator skips invalid children (such as empty values or `__x__`).

Parameters

- **force_create** (Optional[bool]) – if True, will always create new products instead of reuse existing ones
- **silent** (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
Iterator[Any]

Returns
Iterator over the products of the children of the node

pull_named_children(*, *force_create=False*, *silent=None*)

Returns an iterator over the products of the child nodes of `self`, including their keys.

The iterator skips invalid children (such as empty values or `__x__`).

Parameters

- **force_create** (Optional[bool]) – if True, will always create new products instead of reuse existing ones
- **silent** (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
Iterator[Tuple[str, Any]]

Returns
Iterator over the products of the children of the node

pulls(**queries*, *default=<class 'omnibelt.typing.unspecified_argument'>*, *silent=None*, ***kwargs*)

Searches in the config based on the given queries and returns the product of the resulting node. The product is the value that node contains, which can be a primitive, a container, or a component. If the product is a container or component, it will be created if it has not been created yet.

If multiple queries are given, the first one that resolves to a node is used If all the queries fail, the default value will be returned (if given), otherwise a `Search.SearchFailed` will be raised.

Parameters

- ***queries** (str) – list of queries to be resolved
- **default** (Optional[Any]) – value to be returned if all the queries fail
- **silent** (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
Any

Returns
Product of the config node corresponding to the first query that doesn't fail

Raises
Search.SearchFailed – if all the queries fail and no default value is given

push(*addr*, *value*, *overwrite=True*, *silent=None*)

Inserts a new node into the config tree.

Parameters

- **addr** (str) – of the new node (i.e. the key or index)
- **value** (Any) – of the new node
- **overwrite** (bool) – if True, overwrites the existing node if there is one

- **silent** (Optional[bool]) – if True, suppresses the reporter from printing messages

Return type
bool

Returns
True if the node was inserted, False if it was not inserted

Raises
ReadonlyError – if the config is readonly

property reporter: *Reporter*

Returns the reporter associated with this config tree.

Return type
Reporter

search(**queries*, *default=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Creates a search object that can be used to resolve the given queries to find the corresponding node.

Note that this method is usually not called directly, but rather through the top level methods such as `ConfigNode.pull()` or `ConfigNode.peek()`.

Parameters

- ***queries** (str) – list of queries to be resolved
- **default** (Optional[Any]) – if all the queries fail, this value will be returned
- ****kwargs** – additional keyword arguments to be passed to the search object constructor

Return type
Search

Returns
Search object that can be used to traverse the config tree

property settings: OrderedDict

Returns the (global) settings associated with this config tree.

Return type
OrderedDict

silence(*silent=True*)

Convenience method for temporarily setting the silent flag of this config node.

Return type
ContextManager

property silent: bool

Returns whether this config node is in silent mode.

Return type
bool

to_yaml(*stream=None*, *default_flow_style=None*, *sort_keys=True*, ***kwargs*)

Dumps the contents of this config node in YAML format to the specified stream.

Parameters

- **stream** – destination stream
- **default_flow_style** – YAML formatting option (see PyYAML documentation)

- **sort_keys** – YAML formatting option (if True, keys are sorted alphabetically)
- ****kwargs** (Any) – additional keyword arguments to pass to PyYAML's dump function

Return type
None

Returns
None

property trace: *Search* | None

Returns the current search trace associated this config node (generally only used by creators and reporters).

Return type
Optional[*Search*]

update(*update*, *, *clear_product=True*)

Updates the contents of this config node with the contents of another config node.

Parameters

- **update** (*ConfigNode*) – the config node to update from
- **clear_product** (bool) – if True, all references to the products of self and update are removed

Return type
ConfigNode

Returns
The updated config node, self

validate()

Recursively validates the contents of this config node.

Including removing any children with the value _x_ (marked for removal).

class Config_Registry(**args*, *_sister_registry_object=None*, *_sister_registry_cls=None*, ***kwargs*)

Bases: Path_Registry

Registry for config files.

entry_cls

alias of Config_Registry_Entry

__init__(*project*)

export(*config*, *name*, *, *root=None*, *fmt=None*)

Exports the given config to the given path (in yaml format).

Parameters

- **config** (*ConfigNode*) – object to export
- **name** (Union[str, Path]) – of file to export to
- **root** (Optional[Path]) – directory to export to (if not provided, the current working directory is used)
- **fmt** (Optional[str]) – format to export to (if not provided, the extension of the file name is used, and defaults to yaml)

Return type
Path

Returns
The path to which the config was exported

`iterate_configs()`

Iterates over all registered config file entries.

Return type
Iterator[NamedTuple]

Returns
An iterator over all registered config file entries.

`register_config(name, path=None, **kwargs)`

Registers a config file with the given name.

Note: It is generally not recommended to register configs manually, but rather to use the `register_config_dir` method to register all configs in a directory at once.

Parameters

- **name** (Union[str, Path]) – to register the config under
- **path** (Union[str, Path]) – of the config file (if not provided, the provided name is assumed to be a path)
- ****kwargs** – Other arguments to pass to the `Config_Registry.register` method

Return type
NamedTuple

Returns
The entry of the config file that was registered

`register_config_dir(root, *, recursive=True, prefix=None, delimiter=None)`

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

- **root** (Union[str, Path]) – root directory to search through
- **recursive** (Optional[bool]) – search recursively through subdirectories for more config yaml files

- **prefix** (Optional[str]) – prefix for names of configs found herein
- **delimiter** (Optional[str]) – string to merge directories when recursively searching (default /)

Return type
List[NamedTuple]

Returns
A list of all config entries that were registered.

exception UnknownBehaviorError(*term*)

Bases: ValueError

While parsing command-line arguments, a meta config was referenced that was not registered

__init__(*term*)

parse_argv(*argv*, *script_name=<class 'omnibelt.typing.unspecified_argument'>*)

Parses command-line arguments and returns a config object that contains the parsed arguments.

Parameters

- **argv** (Sequence[str]) – raw command-line arguments
- **script_name** (Optional[str]) – if not provided, the script name is inferred from argv

Return type
AbstractConfig

Returns
A config object containing the parsed arguments

Raises

- ***UnknownBehaviorError*** – if an unknown behavior config is encountered
- **ValueError** – if an argument is invalid

find_local_config_entry(*name*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the entry for a config by name.

Parameters

- **name** (str) – used to register the config
- **default** (Optional[Any]) – Default value to return if the artifact is not found.

Return type
NamedTuple

Returns
The entry for the config

Raises
ConfigNotFoundError – if the config is not registered

find_project_config_entry(*name*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the entry for a config by name. Including checking the nonlocal projects.

Parameters

- **name** (str) – of the config file used to register the config
- **default** (Optional[Any]) – Default value to return if the artifact is not found.

Return type
NamedTuple

Returns
The entry for the config

Raises
ConfigNotFoundError – if the config is not registered

find_config_path(*name*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the path to a config file by name. Including checking the nonlocal projects.

Parameters

- **name** (str) – of the config file used to register the config
- **default** (Optional[Any]) – Default value to return if the artifact is not found.

Return type
Path

Returns
The path to the config file

Raises
ConfigNotFoundError – if the config is not registered

load_raw_config(*path*)

Loads raw data of a config file (formats: JSON, YAML, TOML).

Parameters
path (Union[str, Path]) – to the config file

Return type
Union[Dict[str, Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], List[Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], str, int, float, bool, None]

Returns
The raw data of the config file

Raises
ValueError – if the config file is not a valid format

exception ConfigCycleError(*bad*)

Bases: ValueError

Indicates that a cycle in the config bases was detected.

__init__(*bad*)

create_config(*configs=None*, *data=None*, *, *project=<class 'omnibelt.typing.unspecified_argument'>*)

Creates a config object from a list of configs and raw data.

Parameters

- **configs** (Optional[Sequence[Union[str, Path]]]) – names of registered configs or paths to config files to load
- **data** (Union[Dict[str, Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], List[Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], str, int, float, bool, None]) – raw data to merge into the config (e.g. from command line arguments)

- **project** (Optional[*AbstractProject*]) – to associate the resulting config with

Return type
AbstractConfig

Returns
The config object

Raises
ConfigNotFoundError – if a requested config is not registered

configurize(*raw*, ***kwargs*)

Converts raw data into a config object (using the config class of this manager ConfigNode).

Parameters

- **raw** (Union[Dict[str, Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], List[Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], str, int, float, bool, None]) – python data structure to convert (e.g. from a JSON or YAML file)
- ****kwargs** (Any) – additional arguments to pass to the config class constructor

Return type
AbstractConfig

Returns
The config object

merge_configs(**configs*)

Given a list of config objects, merges them into a single config object in the given order.

Parameters
***configs** (*AbstractConfig*) – objects to merge

Return type
AbstractConfig

Returns
The merged config object

static update_config(*base*, *update*)

Updates a config object with the contents of another config object.

Parameters

- **base** (*AbstractConfig*) – config object to update
- **update** (*AbstractConfig*) – config object to merge into the base

Return type
AbstractConfig

Returns
The updated config object

1.25 Mix-ins

class omnifig.mixins.**Activatable**(**args*, ***kwargs*)

Bases: object

Mix-in class for objects that can be activated and deactivated.

Once activated, the object is in a usable state, and cannot be activated again (and the object can only be deactivated if it is already activated).

Primarily used by *omnifig.abstract.AbstractProfile* and *omnifig.abstract.AbstractProject*.

__init__(**args*, ***kwargs*)

property is_activated: bool

Flag whether the object is currently activated

Return type
bool

activate(**args*, ***kwargs*)

Top-level method to activate the object.

Parameters

- ***args** (Any) – Arguments to pass to the _activate() method
- ****kwargs** (Any) – Keyword arguments to pass to the _activate() method

Return type
None

Returns
None

deactivate(**args*, ***kwargs*)

Top-level method to deactivate the object.

Parameters

- ***args** – Arguments to pass to the _deactivate() method
- ****kwargs** – Keyword arguments to pass to the _deactivate() method

Returns
None

class omnifig.mixins.**FileInfo**(*data=None*, ***kwargs*)

Bases: object

Mix-in class for objects that loads and stores information from a file.

static load_raw_info(*path*)

Loads the info yaml file at the given path.

Parameters
path (Path) – File path containing the info yaml file

Return type
Dict[str, Any]

Returns
dict containing the loaded info

`__init__`(*data=None*, ***kwargs*)

Loads the info if the provided *data* is a file path.

Parameters

- **data** (Union[str, Path, Dict[str, Any]]) – A file path to a yaml file, or a dictionary containing the info
- ****kwargs** (Any) – Other arguments passed on to `super()`

`property name`

The name of the object, as specified in the info file.

`extract_info`(*other*)

Extracts the info from the given object and stores it in this object.

Usually used to replace `other` with `self`.

Parameters
other (*FileInfo*) – The source object to extract the info from

Returns
None

1.26 Profiles

`class` omnifig.organization.profiles.`ProfileBase`(*data=None*)

Bases: *AbstractProfile*

Generally, a run environment uses a single profile to keep track of loading projects, and invoking the top level methods (such as *entry()*, *main()*, *run()*, *quick_run()*, etc.).

It is recommended to subclass this class to create a custom profile classes with expected functionality (unlike AbstractProfile).

behavior_registry = {'debug': _Behavior_Registry_Entry(name='debug', cls=<class 'omnifig.behaviors.debug.Debug'>, description='Switch to debug mode'), 'help': _Behavior_Registry_Entry(name='help', cls=<class 'omnifig.behaviors.help.Help'>, description='Display this help message'), 'quiet': _Behavior_Registry_Entry(name='quiet', cls=<class 'omnifig.behaviors.quiet.Quiet'>, description='Set config to silent')}

`Project`

alias of *ProjectBase*

`classmethod get_project_type`(*ident*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Gets the project type entry for the given identifier (from a registry).

Parameters

- **ident** (str) – Name of the registered project type.
- **default** (Optional[Any]) – Default value to return if the identifier is not found.

Return type
NamedTuple

Returns
Project type entry.

classmethod replace_profile(*profile=None*)

Replaces the current profile instance with the given profile. This is used to set the global profile.

Parameters
profile (*ProfileBase*) – New profile instance.

Return type
ProfileBase

Returns
Old profile instance (which is now replaced).

classmethod get_profile()

Gets the current profile instance of the runtime environment.

Return type
ProfileBase

Returns
Profile instance.

classmethod register_behavior(*name*, *typ*, *, *description=None*)

Registers a new behavior in the profile.

Behaviors are classes which are instantiated and managed by .

Parameters

- **name** (str) – Name of the behavior.
- **typ** (Type[*AbstractBehavior*]) – Behavior class (recommended to subclass AbstractBehavior).
- **description** (Optional[str]) – Description of the behavior.

Return type
NamedTuple

Returns
Registration entry for the behavior.

classmethod get_behavior(*name*)

Gets the behavior entry for the given identifier (from the registry).

Parameters
name (str) – Name of the registered behavior.

Return type
NamedTuple

Returns
Behavior entry.

classmethod iterate_behaviors()

Iterates over all registered behaviors.

Return type
Iterator[NamedTuple]

Returns
Iterator over all behavior entries.

__init__(*data=None*)

Loads the info if the provided *data* is a file path.

Parameters

- **data** (Dict[str, Any]) – A file path to a yaml file, or a dictionary containing the info
- ****kwargs** – Other arguments passed on to super()

property path

entry(*script_name=None*)

Primary entry point for the profile. This method is called when using the **fig** command.

Parameters
script_name (Optional[str]) – Manually specified script name to run (if not provided, will be parsed from sys.argv).

Return type
None

Returns
None

initialize(**projects*, ***kwargs*)

Initializes the specified projects (including activating them, which generally registers all associated configs and imports files and packages)

Parameters
***projects** (str) – Identifiers of projects to initialize (activates the current project only, if none is provided).

Return type
None

Returns
None

main(*argv*, *script_name=<class 'omnibelt.typing.unspecified_argument'>*)

Runs the script with the given arguments using *main()* of the current project.

Parameters

- **argv** (Sequence[str]) – List of top-level arguments (expected to be sys.argv[1:]).
- **script_name** (Optional[str]) – specified name of the script
- **object).** *((defaults to what is specified in argv when it is parsed into a config)* –

Return type
None

Returns
The output of the script.

run_script(*script_name*, *config*, **args*, ***kwargs*)

Runs the script registered with the given name and the given arguments using *run_script()* of the current project.

Parameters

- **script_name** (str) – Name of the script to run (must be registered).

- **config** (*AbstractConfig*) – Config object to run the script with.
- ***args** (Any) – Manual arguments to pass to the script.
- ****kwargs** (Any) – Manual keyword arguments to pass to the script.

Return type
Any

Returns
The output of the script.

run(*config*, **args*, ***kwargs*)

Runs the script with the given arguments using *run()* of the current project.

Parameters

- **config** (*AbstractConfig*) – Config object to run the script with.
- ***args** (Any) – Manual arguments to pass to the script.
- ****kwargs** (Any) – Manual keyword arguments to pass to the script.

Returns
The output of the script.

quick_run(*script_name*, **configs*, ***parameters*)

Creates a config object and runs the script using *quick_run()* of the current project.

Parameters

- **script_name** (str) – Name of the script to run (should be registered).
- ***configs** (str) – Names of registered config files to load and merge (in order of precedence).
- ****parameters** (Any) – Manual config parameters to populate the config object with.

Returns
Output of the script.

cleanup(**args*, ***kwargs*)

Calls *cleanup()* of the current project.

Parameters

- ***args** (Any) – Arguments to pass to the cleanup function.
- ****kwargs** (Any) – Keyword arguments to pass to the cleanup function.

Return type
None

Returns
None

create_config(**configs*, ***parameters*)

Process the provided data to create a config object (using the current project).

Parameters

- **configs** (str) – usually a list of parent configs to be merged
- **parameters** (Union[Dict[str, Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], List[Union[Dict[str,

JSONABLE], List[JSONABLE], str, int, float, bool, None]], str, int, float, bool, None]) – any manual parameters to include in the config object

Return type
AbstractConfig

Returns
Config object resulting from loading/merging *configs* and including *data*.

parse_argv(*argv*, *script_name=None*)

Parses the given arguments and returns a config object.

Arguments are expected in the following order (all of which are optional):

1. Behaviors to modify the config loading process and script execution.
2. Name of the script to run.
3. Names of registered config files that should be loaded and merged (in order of precedence).
4. Manual config parameters (usually keys, prefixed by `--` and corresponding values)

Parameters

- **argv** (Sequence[str]) – List of arguments to parse (expected to be `sys.argv[1:]`).
- **script_name** – Manually specified name of the script (defaults to what is specified in the resulting config).

Return type
AbstractConfig

Returns
Config object containing the parsed arguments.

extract_info(*other*)

Extract data from the provided profile instance and store it in self.

Recommended to use if a project expects a custom profile different from the currently used one.

Parameters
profile – Base profile instance.

Return type
None

Returns
None

get_current_project()

Gets the current project instance.

Return type
ProjectBase

Returns
Current project instance.

create_project(*name*, *path=None*, *set_as_current=True*)

Creates a new project instance and registers it.

Parameters

- **name** (str) – Name of the project.

- **path** (Path) – Path to the project (defaults to the current working directory).
- **set_as_current** (bool) – If True, sets the new project as the current project.

Return type
AbstractProject

Returns
New project instance.

switch_project(*ident=None*)

Switches the current project to the one with the given identifier.

Parameters
ident (Union[str, *AbstractProject*]) – Name of the project to switch to, defaults to the default project (with name: None).

Return type
AbstractProject

Returns
New current project instance.

iterate_projects()

Iterates over all loaded projects.

Return type
Iterator[*AbstractProject*]

Returns
Iterator over all loaded project instances.

project_context(*ident=None*)

Context manager to temporarily switch to a different current project.

Parameters
ident (Union[str, *AbstractProject*]) – Name of the project to switch to, defaults to the default project (with name: None).

Return type
ContextManager[*AbstractProject*]

Returns
Context manager to switch to the specified project.

omnifig.organization.profiles.**get_profile**()

Returns the current profile instance.

Return type
ProfileBase

1.27 Project Base

class omnifig.organization.workspaces.**ProjectBase**(*path=None*, *profile=None*, ***kwargs*)

Bases: *AbstractProject*

Generally, each workspace (e.g. repo, directory, package) should define its own project, which keeps track of all the configs and artifacts defined therein.

Projects don't just contain all the project specific registries, but can also modify the behavior of the top-level methods such as (such as `main()`, `run()`, etc.).

It is recommended to subclass this class to create a custom project class with expected functionality (unlike `AbstractProject`) and automatically register it to the global project type registry.

global_type_registry = {'default': Class_Registry_Entry(name='default', cls=<class 'omnifig.organization.default.Project'>), 'general': Class_Registry_Entry(name='general', cls=<class 'omnifig.organization.workspaces.GeneralProject'>)}

Global registry for project types (usually used by the profile to create custom project types).

classmethod get_project_type(*ident*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Accesses the project type entry for the given identifier (from a registry).

Return type
`NamedTuple`

class omnifig.organization.workspaces.**GeneralProject**(*path*, ***, *script_registry=None*, *config_manager=None*, ***kwargs*)

Bases: *ProjectBase*

Project class that includes basic functionality such as a script registry and config manager.

info_file_names = {'.fig.project.yaml', '.fig.project.yml', '.omnifig.yaml', '.omnifig.yml', 'fig.project.yaml', 'fig.project.yml', 'omnifig.yaml', 'omnifig.yml'}

infer_path(*path=None*)

Infers the path of the project from the given path or the current working directory.

Return type
`Path`

xray(*artifact*, ***, *sort=False*, *reverse=False*, *as_list=False*)

Prints a list of all artifacts of the given type accessible from this project (including related and active base projects).

Parameters

- **artifact** (`str`) – artifact type (e.g. 'script', 'config')
- **sort** (`Optional[bool]`) – sort the list of artifacts by name
- **reverse** (`Optional[bool]`) – reverse the order of the list of artifacts
- **as_list** (`Optional[bool]`) – instead of printing, return the list of artifacts

Returns
if as_list is True, returns a list of artifacts

Return type
list

Raises
UnknownArtifactTypeError – if the given artifact type does not exist

class Script_Registry(**args*, *_sister_registry_object=None*, *_sister_registry_cls=None*, ***kwargs*)

Bases: Function_Registry

Registry for scripts (functions) that can be run from the command line.

entry_cls

alias of Script_Registry_Entry

Config_Manager

alias of *ConfigManager*

__init__(*path*, ***, *script_registry=None*, *config_manager=None*, ***kwargs*)

Loads the info if the provided *data* is a file path.

Parameters

- **data** – A file path to a yaml file, or a dictionary containing the info
- ****kwargs** – Other arguments passed on to super()

load()

Loads all dependencies for the project including config files/directories, packages, and source files based on the contents of the project's info file.

Packages are imported (or reloaded) and source files are executed locally.

Returns
self

Return type
Project

unload()

Unloads all dependencies for the project (opposite of *load()*).

load_configs(*paths=()*)

Registers all specified config files and directories

Return type
None

extract_info(*other*)

Extracts the info from the given project into this project.

Return type
None

validate()

Validates the project and returns a new project with the validated info.

Return type
AbstractProject

property name

The name of the project

property root: Path

The root directory of the project.

Return type
Path

property info_path: Path

The path to the info file.

Return type
Path

create_config(**parents*, ***parameters*)

Creates a config with the given parameters using the config manager.

parse_argv(*argv*, *, *script_name=<class 'omnibelt.typing.unspecified_argument'>*)

Parses the given command line arguments into a config object.

Return type
AbstractConfig

behaviors()

Iterates over all behaviors associated with this project.

Return type
Iterator[*AbstractBehavior*]

validate_run(*config*)

Validates the project `self` using the given config object before running it.

More specifically, this method calls `validate_project()` on all behaviors associated with the project with the config object as argument. If any of the behaviors returns a new project, this project is returned (and used for the script execution instead). Otherwise, `None` is returned.

Parameters
config (*AbstractConfig*) – The config object to use for validation.

Return type
Optional[*AbstractProject*]

Returns
The new project to use for the script execution or `None` if no new project was returned.

main(*argv*, *script_name=<class 'omnibelt.typing.unspecified_argument'>*)

Runs the script with the given arguments using the config object obtained by parsing `argv`.

More specifically, this method does the following:

1. Activates the project (loads any specified source files or packages for the script, if not done yet).
2. Instantiates all behaviors associated with the project.
3. Parses the command line arguments into a config object.
4. Validates the current project using the config object and behaviors (see *validate_run()*).
5. Runs the script using the config object (see *run_script()*).
6. Cleans up the project (see `cleanup()`).

Parameters

- **argv** (Sequence[str]) – List of top-level arguments (expected to be `sys.argv[1:]`).

- **script_name** (Optional[str]) – specified name of the script
- **object).** ((*defaults to what is specified in argv when it is parsed into a config*) –

Return type
Any

Returns
The output of the script.

run_script(*script_name*, *config*, **args*, ***kwargs*)

Runs the script with the given arguments using *run()* of the current project.

Parameters

- **script_name** (str) – The script name to run (must be registered).
- **config** (*AbstractConfig*) – Config object to run the script with (must include the script under _meta.script_name).
- **args** (Any) – Additional positional arguments to pass to the script.
- **kwargs** (Any) – Additional keyword arguments to pass to the script.

Return type
Any

Returns
The output of the script.

exception NoScriptError

Bases: ValueError

Raised when the script name is not specified.

run(*config*, **args*, ***kwargs*)

Runs the given script with the given config using this project.

Parameters

- **config** (*AbstractConfig*) – The config to use for running the script (must include the script under _meta.script_name).
- **args** (Any) – Additional positional arguments to pass to the script.
- **kwargs** (Any) – Additional keyword arguments to pass to the script.

Return type
Any

Returns
The return value of the script.

exception TerminationFlag(*out=None*)

Bases: KeyboardInterrupt

Raised if the subsequent script should not be run.

__init__(*out=None*)

Prevents the subsequent script from being run.

Parameters
out (Any) – User-defined output to be returned instead of the output of the script.

out = None

exception IgnoreException(*out=None*)

Bases: Exception

Raised if the underlying exception raised while running the script should be ignored.

__init__(*out=None*)

Instead of raising the exception, out will be returned.

Parameters
out (Any) – User-defined output to be returned instead of raising the exception.

exception UnknownArtifactTypeError

Bases: KeyError

Raised when an unknown artifact type is encountered.

find_local_artifact(*artifact_type*, *ident*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the artifact of the given type and registered with the given identifier.

Parameters

- **artifact_type** (str) – The type of artifact to find.
- **ident** (str) – The identifier of the artifact to find.
- **default** (Optional[Any]) – The default value to return if the artifact is not found.

Return type
Optional[NamedTuple]

Returns
The artifact entry from the registry corresponding to the given type.

Raises

- *UnknownArtifactTypeError* – If the artifact type is not registered.
- *UnknownArtifactError* – If the artifact is not found and no default is given.

find_artifact(*artifact_type*, *ident*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds an artifact in the project's registries, including related and active base projects. Artifacts are data or functionality such as configs and components.

Parameters

- **artifact_type** (str) – Type of artifact to find (eg. 'config' or 'component').
- **ident** (str) – Name of the artifact that was registered.
- **default** (Optional[Any]) – Default value to return if the artifact is not found.

Return type
NamedTuple

Returns
Artifact object, or, if a default value is given and artifact is not found.

Raises
UnknownArtifactError – if the artifact is not found and no default is specified.

register_artifact(*artifact_type*, *ident*, *artifact*, *, *project=<class 'omnibelt.typing.unspecified_argument'>*, ***kwargs*)

Registers the given artifact with the given type and identifier and any additional info.

Parameters

- **artifact_type** – The type of artifact to register.
- **ident** (str) – The identifier of the artifact to register.
- **artifact** (Union[str, Type, Callable]) – The artifact to register (usually a type or config file path).
- **project** (Optional[*AbstractProject*]) – The project to register the artifact for (defaults to this project).
- **kwargs** – Additional keyword arguments to pass to the registry.

Return type
NamedTuple

Returns
The artifact entry from the registry corresponding to the given type.

Raises
UnknownArtifactTypeError – If the artifact type does not exist.

iterate_artifacts(*artifact_type*)

Iterates over all artifacts of the given type.

Parameters
artifact_type (str) – The type of artifact to iterate over (e.g. 'script', 'config').

Return type
Iterator[NamedTuple]

Returns
An iterator over all artifacts of the given type.

Raises
UnknownArtifactTypeError – If the artifact type does not exist.

find_config(*name*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the config with the given name.

Parameters

- **name** (str) – name the config was registered with.
- **default** (Optional[Any]) – default value to return if the config is not found.

Return type
NamedTuple

Returns
The config entry corresponding to the given name.

Raises
UnknownArtifactError – If the config is not found and no default is given.

register_config(*name*, *path=None*, ***kwargs*)

Registers a config file with the given name.

Note: It is generally not recommended to register configs manually, but rather to use the `register_config_dir` method to register all configs in a directory at once.

Parameters

- **name** (`Union[str, Path]`) – to register the config under
- **path** (`Union[str, Path]`) – of the config file (if not provided, the provided name is assumed to be a path)
- ****kwargs** – Other arguments to pass to the `Path_Registry.register` method

Return type
`NamedTuple`

Returns
The entry of the config file that was registered

`iterate_configs()`

Iterates over all registered config file entries.

Return type
`Iterator[NamedTuple]`

Returns
An iterator over all registered config file entries.

`register_config_dir(`*path*, *, *recursive=True*, *prefix=None*, *delimiter=None*`)`

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** (`Union[str, Path]`) – path to root directory to search through
- **recursive** (`Optional[bool]`) – search recursively through subdirectories for more config yaml files
- **prefix** (`Optional[str]`) – prefix for names of configs found herein
- **delimiter** (`Optional[str]`) – string to merge directories when recursively searching (default `/`)

Return type
`List[NamedTuple]`

Returns
A list of all config entries that were registered.

`find_script`(*name*, *default=<class 'omnibelt.typing.unspecified_argument'>*)

Finds the script with the given name.

Parameters

- **name** (`str`) – the script was registered with.
- **default** (`Optional[Any]`) – default value to return if the script is not found.

Return type
`NamedTuple`

Returns
The script entry corresponding to the given name.

Raises
UnknownArtifactError – If the script is not found and no default is given.

`register_script`(*name*, *fn*, ***, *description=None*, *hidden=None*)

Register a script with the given name.

Parameters

- **name** (`str`) – to register the script under
- **fn** (`Callable[[AbstractConfig], Any]`) – the script function (should expect the first positional argument to be the config object)
- **description** (`Optional[str]`) – description of the script
- **hidden** (`Optional[bool]`) – whether to hide the script from the list of available scripts
- **underscore)** (*(defaults to whether the name starts with an)* –

Return type
`NamedTuple`

Returns
The entry of the script that was registered

`iterate_scripts`()

Iterates over all registered script entries.

Return type
`Iterator[NamedTuple]`

Returns
An iterator over all registered script entries.

1.28 Registration

class omnifig.registration.**script**(*name=None*, *description=None*, *, *hidden=None*)

Bases: _Project_Registration_Decorator

Decorator to register a script.

Scripts are callable objects (usually functions) with only one input argument (the config object) and can be called from the command line using the `fig` command.

__init__(*name=None*, *description=None*, *, *hidden=None*)

Parameters

- **name** (Optional[str]) – name of item to be registered (defaults to its __name__)
- **description** (Optional[str]) – a short description of what the script does (defaults to first line of its docstring)
- **hidden** (bool) – if True, the script will not be listed in the help menu

static register_project(*project*, *name*, *item*, *description=None*, *hidden=None*, ***kwargs*)

Must be implemented by subclasses to register the item with the current project

Return type
None

class omnifig.registration.**creator**(*name=None*, *description=None*)

Bases: _Project_Registration_Decorator

Decorator to register a creator.

Creators are generally subclasses of `AbstractCreator` and are used to create objects from the config.

Usually, the default creator is sufficient, but this decorator can be used to register a custom creator.

__init__(*name=None*, *description=None*)

Parameters

- **name** (Optional[str]) – name of item to be registered (defaults to its __name__)
- **description** (Optional[str]) – a short description of what the script does (defaults to first line of its docstring)

static register_project(*project*, *name*, *item*, *description=None*, ***kwargs*)

Must be implemented by subclasses to register the item with the current project

Return type
None

class omnifig.registration.**component**(*name=None*, *description=None*, *creator=None*)

Bases: _Project_Registration_Decorator

Decorator to register a component.

Components are (usually) classes, and can be automatically be instantiated from the config object (using the `_type` key).

There are generally two different ways to use components. Both use a creator (see `AbstractCreator`):

1. **If the component is a subclass of `Configurable`,**
   arguments in __init__ can be automatically be filled in with the config object.

2. **Otherwise, the component will be instantiated (by default) with the following signature:** `config, *args, **kwargs`, where `config` is the config object, while `*args` and `**kwargs` are arguments manually passed to the creator. This is the signature expected for `init_from_config()` if the component is a subclass of `AbstractConfigurable` and *__init__()* otherwise.

__init__(*name=None*, *description=None*, *creator=None*)

Decorator to register a component.

Parameters

- **name** (Optional[str]) – name of item to be registered (defaults to its __name__)
- **description** (Optional[str]) – a short description of what the script does (defaults to first line of its docstring)
- **creator** (Optional[str]) – name of the creator that should be used to create this component (generally not recommended)

static register_project(*project*, *name*, *item*, *description=None*, *creator=None*, ***kwargs*)

Must be implemented by subclasses to register the item with the current project

Return type
None

class omnifig.registration.**modifier**(*name=None*, *description=None*)

Bases: _Project_Registration_Decorator

Decorator to register a modifier.

Modifiers are "runtime mixins" for components and must be classes. When specifying a component to be modified with the _mod key in the config, a new type is dynamically created for which the bases are all the specified modifiers followed by the original component.

__init__(*name=None*, *description=None*)

Decorator to register a modifier.

Parameters

- **name** (Optional[str]) – name of item to be registered (defaults to its __name__)
- **description** (Optional[str]) – a short description of what the script does (defaults to first line of its docstring)

static register_project(*project*, *name*, *item*, *description=None*, ***kwargs*)

Must be implemented by subclasses to register the item with the current project

Return type
None

class omnifig.registration.**autoscript**(*name=None*, *description=None*, *aliases=None*, ***kwargs*)

Bases: _AutofillMixin, *script*

Convienence decorator to register scripts where the arguments of the script signature are automatically extracted from the config before running the script.

Note: This is generally only recommended for simple, short scripts (since it severely limits the usage of the config object by the script).

__init__(*name=None*, *description=None*, *aliases=None*, ***kwargs*)

Decorator to register a script (where arguments are extracted from the config automatically).

Parameters

- **name** (Optional[str]) – name of item to be registered (defaults to its __name__)
- **description** (Optional[str]) – a short description of what the script does (defaults to first line of its docstring)
- **aliases** (Optional[Dict[str, Union[str, Sequence[str]]]]) – alternative names for arguments (can have multiple aliases per argument)

class omnifig.registration.**autocomponent**(*name=None*, *description=None*, *aliases=None*, *creator=None*)

Bases: _AutofillMixin, *component*

Convienence decorator to register components where the arguments of the component function are automatically extracted from the config

---

Note: This is generally only recommended for simple components that are functions (rather than classes), since class components should simply subclass Configurable for effectively the same behavior.

__init__(*name=None*, *description=None*, *aliases=None*, *creator=None*)

Decorator to register a component (where arguments are extracted from the config automatically).

Parameters

- **name** (Optional[str]) – name of item to be registered (defaults to its __name__)
- **description** (Optional[str]) – a short description of what the script does (defaults to first line of its docstring)
- **aliases** (Optional[Dict[str, Union[str, Sequence[str]]]]) – alternative names for arguments (can have multiple aliases per argument)
- **creator** (Union[str, *AbstractCreator*, None]) – name of the creator that should be used to create this component

1.29 Top-level Interface

omnifig.top.**get_current_project**()

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

Return type
AbstractProject

omnifig.top.**get_project**(*ident=None*)

Checks the profile to return (and possibly load) a project given the name or path ident

Return type
AbstractProject

omnifig.top.**switch_project**(*ident=None*)

Switches the current project to the one of thegiven the project name or path ident

Return type
AbstractProject

omnifig.top.**iterate_projects**()

Iterate over all loaded projects

Return type
Iterator[*AbstractProject*]

omnifig.top.**project_context**(*ident=None*)

Context manager for switching to a project and then switching back

Return type
ContextManager

omnifig.top.**entry**(*script_name=<class 'omnibelt.typing.unspecified_argument'>*)

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 (Optional[str]) – script to be run (maybe set with arguments) (overrides other arguments if provided)

Return type
None

Returns
None

omnifig.top.**main**(*argv*, *script_name=<class 'omnibelt.typing.unspecified_argument'>*)

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** (Sequence[str]) – raw arguments as if passed in through the terminal
- **script_name** (Optional[str]) – name of registered script to be run (maybe set with arguments) (overrides other arguments if provided)

Return type
Any

Returns
The output of script that is run

omnifig.top.**run_script**(*script_name*, *config*, **args*, ***kwargs*)

Runs the specified script registered with `script_name` using the current project.

Parameters

- **script_name** (str) – Must be registered in the current project
- **config** (*AbstractConfig*) – The config object passed to the script
- ***args** (Any) – Manual arguments to be passed to the script
- ****kwargs** (Any) – Manual keyword arguments to be passed to the script

Return type
Any

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

omnifig.top.**run**(*config*, **args*, ***kwargs*)

Runs the specified script registered with `script_name` using the current project.

Parameters

- **config** (*AbstractConfig*) – The config object passed to the script
- ***args** (Any) – Manual arguments to be passed to the script
- ****kwargs** (Any) – Manual keyword arguments to be passed to the script

Return type
Any

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

omnifig.top.**quick_run**(*script_name*, **parents*, ***parameters*)

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

Parameters

- **script_name** (str) – name of registered script that is to be run
- ***parents** (str) – any names of registered configs to load
- ****parameters** (Union[Dict[str, Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], List[Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], str, int, float, bool, None]) – any additional arguments to be provided manually

Return type
Any

Returns
The script output

omnifig.top.**initialize**(**projects*, ***settings*)

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** (str) – additional projects that should be initialized
- **settings** (Any) – extra global settings (unused by default)

Return type
None

Returns
None

omnifig.top.**create_config**(**configs*, ***parameters*)

Process the provided data to create a config object (using the current project).

Parameters

- **configs** (str) – usually a list of parent configs to be merged
- **parameters** (Union[Dict[str, Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], List[Union[Dict[str, JSONABLE], List[JSONABLE], str, int, float, bool, None]], str, int, float, bool, None]) – any manual parameters to include in the config object

Return type
AbstractConfig

Returns
Config object resulting from loading/merging *configs* and including *data*.

omnifig.top.**parse_argv**(*argv*, *, *script_name=None*)

Parses the given arguments and returns a config object.

Arguments are expected in the following order (all of which are optional):

1. Meta rules to modify the config loading process and run mode.
2. Name of the script to run.
3. Names of registered config files that should be loaded and merged (in order of precedence).
4. Manual config parameters (usually keys, prefixed by `--` and corresponding values)

Parameters

- **argv** (Sequence[str]) – List of arguments to parse (expected to be `sys.argv[1:]`).
- **script_name** (Optional[str]) – Manually specified name of the script (defaults to what is specified in the resulting config).

Return type
AbstractConfig

Returns
Config object containing the parsed arguments.

CHAPTER

TWO

CITATIONS

If you used `omni-fig` in your work, please cite it using:

```
@misc{leeb2022omnifig,
  title = {Omni-fig: Unleashing Project Configuration and Organization in Python},
  author = {Leeb, Felix},
  publisher = {GitHub},
  year = {2022}
}
```

PYTHON MODULE INDEX

O

INDEX

Symbols

A

B

C

D

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

X