
ml*collections*

Release 0.1.0

Apr 21, 2023

1	ml_collections.config_dict examples	3
2	ml_collections.config_flags examples	5
3	ml_collections.config_dict package	7
4	ml_collections.config_flags package	19
5	How to Contribute	25
6	Testing	27
	Python Module Index	29
	Index	31

ML Collections is a library of Python Collections designed for ML use cases.

CHAPTER 1

`ml_collections.config_dict` examples

CHAPTER 2

ml_collections.config_flags examples

ml_collections.config_dict package

Classes for defining configurations of experiments and models.

3.1 ConfigDict class

class ml_collections.config_dict.**ConfigDict** (*initial_dictionary: Optional[Mapping[str, Any]] = None, type_safe: bool = True, convert_dict: bool = True*)

Base class for configuration objects used in DeepMind.

This is a container for configurations. It behaves similarly to Lua tables. Specifically:

- it has dot-based access as well as dict-style key access,
- it is type safe (once a value is set one cannot change its type).

Typical usage example:

```
from ml_collections import config_dict

cfg = config_dict.ConfigDict()
cfg.float_field = 12.6
cfg.integer_field = 123
cfg.another_integer_field = 234
cfg.nested = config_dict.ConfigDict()
cfg.nested.string_field = 'tom'

print(cfg)
```

Config dictionaries can also be used to pass named arguments to functions:

```
from ml_collections import config_dict

def print_point(x, y):
```

(continues on next page)

(continued from previous page)

```

print "{}, {}".format(x, y)

point = config_dict.ConfigDict()
point.x = 1
point.y = 2
print_point(**point)

```

Note that, depending on your use case, it may be easier to use the `create` function in this package to construct a `ConfigDict`:

```
from ml_collections import config_dict point = config_dict.create(x=1, y=2)
```

Differently from standard *dicts*, *ConfigDicts* also have the nice property that iterating over them is deterministic, in a fashion similar to *collections.OrderedDicts*.

__init__ (*initial_dictionary: Optional[Mapping[str, Any]] = None, type_safe: bool = True, convert_dict: bool = True*)
Creates an instance of `ConfigDict`.

Warning: In most cases, this faithfully reproduces the reference structure of `initial_dictionary`, even if `initial_dictionary` is self-referencing. However, unexpected behavior occurs if self-references are contained within list, tuple, or custom types. For example:

```

d = {}
d['a'] = d
d['b'] = [d]
cd = ConfigDict(d)
cd.a    # refers to cd, type ConfigDict. Expected behavior.
cd.b    # refers to d, type dict. Unexpected behavior.

```

Warning: `FieldReference` values may be changed. If `initial_dictionary` contains a `FieldReference` with a value of type `dict` or `FrozenConfigDict`, that value is converted to `ConfigDict`.

Parameters

- **initial_dictionary** – May be one of the following:
 - 1) `dict`. In this case, all values of `initial_dictionary` that are dictionaries are also be converted to `ConfigDict`. However, dictionaries within values of non-dict type are untouched.
 - 2) `ConfigDict`. In this case, all attributes are uncopied, and only the top-level object (self) is re-addressed. This is the same behavior as Python `dict`, `list`, and `tuple`.
 - 3) `FrozenConfigDict`. In this case, `initial_dictionary` is converted to a `ConfigDict` version of the initial dictionary for the `FrozenConfigDict` (reversing any mutability changes `FrozenConfigDict` made).
- **type_safe** – If set to `True`, once an attribute value is assigned, its type cannot be overridden without `.ignore_type()` context manager (default: `True`).
- **convert_dict** – If set to `True`, all `dict` used as value in the `ConfigDict` will automatically be converted to `ConfigDict` (default: `True`).

convert_dict

Returns `True` if it is converting `dicts` to `ConfigDict` automatically.

copy_and_resolve_references (*visit_map=None*)

Returns a `ConfigDict` copy with `FieldReferences` replaced by values.

If the object is a `FrozenConfigDict`, the copy returned is also a `FrozenConfigDict`. However, note that `FrozenConfigDict` should already have `FieldReferences` resolved to values, so this method effectively produces a deep copy.

Note: As with `__eq__()` and `__init__()`, this may not behave as expected on a `ConfigDict` with self-references contained in lists, tuples, or custom types.

Parameters `visit_map` – A mapping from `ConfigDict` object ids to their copy. Method is recursive in nature, and it will call “.copy_and_resolve_references(visit_map)” on each encountered object, unless it is already in `visit_map`.

Returns `ConfigDict` copy with previous `FieldReferences` replaced by values.

eq_as_configdict (*other*)

Type-invariant equals.

This is like `__eq__`, except it does not distinguish `FrozenConfigDict` from `ConfigDict`. For example:

```
cd = ConfigDict()
fcd = FrozenConfigDict()
fcd.eq_as_configdict(cd) # Returns True
```

Parameters `other` – Object to compare self to.

Returns `True` if `self == other` after conversion to `ConfigDict`.

Return type same

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

Returns value if key is present, or a user defined value otherwise.

get_oneway_ref (*key*)

Returns a one-way `FieldReference`.

Example:

```
cfg = config_dict.ConfigDict(dict(a=1))
cfg.b = cfg.get_oneway_ref('a')

cfg.a = 2
print(cfg.b) # 2

cfg.b = 3
print(cfg.a) # 2 (would have been 3 if using get_ref())
print(cfg.b) # 3
```

Parameters `key` – Key for field we want to reference.

get_type (*key*)

Returns type of the field associated with a key.

ignore_type ()

Context manager which temporarily turns off type safety recursively.

is_locked

Returns `True` if object is locked.

is_type_safe

Returns `True` if config dict is type safe.

items (*preserve_field_references=False*)

Returns list of dictionary key, value pairs, sorted by key.

Parameters **preserve_field_references** – (bool) Whether to preserve FieldReferences if the ConfigDict has them. By default, False: any FieldReferences will be resolved in the result.

Returns The key, value pairs in the config, sorted by key.

iteritems (*preserve_field_references=False*)

Deterministically iterates over dictionary key, value pairs.

Parameters **preserve_field_references** – (bool) Whether to preserve FieldReferences if the ConfigDict has them. By default, False: any FieldReferences will be resolved in the result.

Yields The key, value pairs in the config, sorted by key.

iterkeys ()

Deterministically iterates over dictionary keys, in sorted order.

itervalues (*preserve_field_references=False*)

Deterministically iterates over values in a config, sorted by their keys.

Parameters **preserve_field_references** – (bool) Whether to preserve FieldReferences if the ConfigDict has them. By default, False: any FieldReferences will be resolved in the result.

Yields The values in the config, sorted by their corresponding keys.

keys ()

Returns the sorted list of all the keys defined in a config.

lock () → ml_collections.config_dict.config_dict.ConfigDict

Locks object, preventing user from adding new fields.

Returns self

to_dict (*visit_map=None, preserve_field_references=False*)

Converts ConfigDict to regular dict recursively with valid references.

By default, the output dict will not contain FieldReferences, any present in the ConfigDict will be resolved. However, if *preserve_field_references* is True, the output dict will contain FieldReferences where the original ConfigDict has them. They will not be the same as the ConfigDict's, and their ops will be applied and dropped.

Note: As with `__eq__()` and `__init__()`, this may not behave as expected on a ConfigDict with self-references contained in lists, tuples, or custom types.

Parameters

- **visit_map** – A mapping from object ids to their dict representation. Method is recursive in nature, and it will call “.to_dict(visit_map)” on each encountered object, unless it is already in visit_map.
- **preserve_field_references** – (bool) Whether the output dict should have FieldReferences if the ConfigDict has them. By default, False: any FieldReferences will be resolved and the result will go to the dict.

Returns

Dictionary with the same values and references structure as a calling ConfigDict.

to_json (*json_encoder_cls=None, **kwargs*)

Returns a JSON representation of the object, fails if there is a cycle.

Parameters

- **json_encoder_cls** – An optional JSON encoder class to customize JSON serialization.
- ****kwargs** – Keyword arguments for `json.dumps`. They cannot contain “cls” as this method specifies it on its own.

Returns JSON representation of the object.

Raises `TypeError` – If self contains set, frozenset, custom type fields or any other objects that are not JSON serializable.

to_json_best_effort (***kwargs*)

Returns a best effort JSON representation of the object.

Tries to serialize objects not inherently supported by JSON encoder. This may result in the `ConfigDict` being partially serialized, skipping the unserializable bits. Ensures that no errors are thrown. Fails if there is a cycle.

Parameters ****kwargs** – Keyword arguments for `json.dumps`. They cannot contain “cls” as this method specifies it on its own.

Returns JSON representation of the object.

to_yaml (***kwargs*)

Returns a YAML representation of the object.

`ConfigDict` serializes types of fields as well as the values of fields themselves. Deserializing the YAML representation hence requires using YAML’s `UnsafeLoader`:

```
` yaml.load(cfg.to_yaml(), Loader=yaml.UnsafeLoader) `
```

or equivalently:

```
` yaml.unsafe_load(cfg.to_yaml()) `
```

Please see the PyYAML documentation and <https://msg.pyyaml.org/load> for more details on the consequences of this.

Parameters ****kwargs** – Keyword arguments for `yaml.dump`.

Returns YAML representation of the object.

unlock () → `ml_collections.config_dict.config_dict.ConfigDict`

Grants user the ability to add new fields to `ConfigDict`.

In most cases, the `unlocked()` context manager should be preferred to the direct use of the `unlock` method.

Returns `self`

unlocked ()

Context manager which temporarily unlocks a `ConfigDict`.

update (**other, **kwargs*)

Update values based on matching keys in another dict-like object.

Mimics the built-in dict’s `update` method: iterates over a given mapping object and adds/overwrites keys with the given mapping’s values for those keys.

Differs from `dict.update` in that it operates recursively on existing keys that are already a `ConfigDict` (i.e. calls their `update()` on the corresponding value from `other`), and respects the `ConfigDict`’s type safety status.

If keyword arguments are specified, the `ConfigDict` is updated with those key/value pairs.

Parameters

- ***other** – A (single) dict-like container, e.g. a dict or `ConfigDict`.
- ****kwargs** – Additional keyword arguments to update the `ConfigDict`.

Raises `TypeError` – if more than one value for *other* is specified.

update_from_flattened_dict (*flattened_dict*, *strip_prefix=""*)

In-place updates values taken from a flattened dict.

This allows a potentially nested source `ConfigDict` of the following form:

```
cfg = ConfigDict({
    'a': 1,
    'b': {
        'c': {
            'd': 2
        }
    }
})
```

to be updated from a dict containing paths navigating to child items, of the following form:

```
updates = {
    'a': 2,
    'b.c.d': 3
}
```

This filters *paths_dict* to only contain paths starting with *strip_prefix* and strips the prefix when applying the update.

For example, consider we have the following values returned as flags:

```
flags = {
    'flag1': x,
    'flag2': y,
    'config': 'some_file.py',
    'config.a.b': 1,
    'config.a.c': 2
}

config = ConfigDict({
    'a': {
        'b': 0,
        'c': 0
    }
})

config.update_from_flattened_dict(flags, 'config.')
```

Then we will now have:

```
config = ConfigDict({
    'a': {
        'b': 1,
        'c': 2
    }
})
```

Parameters

- **flattened_dict** – A mapping (key path) -> value.
- **strip_prefix** – A prefix to be stripped from *path*. If specified, only paths matching *strip_prefix* will be processed.

Raises `KeyError` – if any of the key paths can't be found.

values (*preserve_field_references=False*)

Returns the list of all values in a config, sorted by their keys.

Parameters **preserve_field_references** – (bool) Whether to preserve `FieldReferences` if the `ConfigDict` has them. By default, `False`: any `FieldReferences` will be resolved in the result.

Returns The values in the config, sorted by their corresponding keys.

3.2 FrozenConfigDict class

class `ml_collections.config_dict.FrozenConfigDict` (*initial_dictionary=None*,
type_safe=True)

Immutable and hashable type of `ConfigDict`.

See `ConfigDict()` documentation above for details and usage.

`FrozenConfigDict` is fully immutable. It contains no lists or sets (at initialization, lists and sets are converted to tuples and frozensets). The only potential sources of mutability are attributes with custom types, which are not touched.

It is recommended to convert a `ConfigDict` to `FrozenConfigDict` after construction if possible.

__init__ (*initial_dictionary=None*, *type_safe=True*)

Creates an instance of `FrozenConfigDict`.

Lists and sets are copied into tuples and frozensets. However, copying is kept to a minimum so tuples, frozensets, and other immutable types are not copied unless they contain mutable types.

Prohibited `initial_dictionary` structures: `initial_dictionary` may not contain any lists or tuples with dictionary, `ConfigDict`, or `FieldReference` elements, or else an error is raised at initialization. It also may not contain loops in the reference structure, i.e. the reference structure must be a Directed Acyclic Graph. This includes loops in list-element and tuple-element references. `initial_dictionary`'s reference structure need not be a tree.

Warning: Unexpected behavior may occur with types other than Python's built-in types. See `ConfigDict()` documentation for details.

Warning: As with `ConfigDict`, `FieldReference` values may be changed. If `initial_dictionary` contains a `FieldReference` with a value of type `dict` or `ConfigDict`, that value will be converted to `FrozenConfigDict`.

Parameters

- **initial_dictionary** – May be one of the following:
 - 1) `dict`. In this case all values of `initial_dictionary` that are dictionaries are also converted to `FrozenConfigDict`. If there are dictionaries contained in lists or tuples, an error is raised.
 - 2) `ConfigDict`. In this case all `ConfigDict` attributes are also converted to `FrozenConfigDict`.

3) FrozenConfigDict. In this case all attributes are uncopied, and only the top-level object (self) is re-addressed.

- **type_safe** – See ConfigDict documentation. Note that this only matters if the FrozenConfigDict is converted to ConfigDict at some point.

3.3 FieldReference class

class ml_collections.config_dict.**FieldReference** (*default*, *field_type=None*, *op=None*, *required=False*)

Reference to a configuration element.

Typed configuration element that can take a None default value. Example:

```
from ml_collections import config_dict

cfg_field = config_dict.FieldReference(0)
cfg = config_dict.ConfigDict({
    'optional': configdict.FieldReference(None, field_type=str)
    'field': cfg_field,
    'nested': {'field': cfg_field}
})

with self.assertRaises(TypeError):
    cfg.optional = 10 # Raises an error because it's defined as an
                    # intfield.

cfg.field = 1 # Changes the value of both cfg.field and cfg.nested.field.
print(cfg)
```

This class also supports lazy computation. Example:

```
ref = config_dict.FieldReference(0)

# Using ref in a standard operation returns another FieldReference. The new
# reference ref_plus_ten will evaluate ref's value only when we call
# ref_plus_ten.get()
ref_plus_ten = ref + 10

ref.set(3) # change ref's value
print(ref_plus_ten.get()) # Prints 13 because ref's value is 3

ref.set(-2) # change ref's value again
print(ref_plus_ten.get()) # Prints 8 because ref's value is -2
```

__init__ (*default*, *field_type=None*, *op=None*, *required=False*)

Creates an instance of FieldReference.

Parameters

- **default** – Default value.
- **field_type** – Type for the values contained by the configuration element. If None the type will be inferred from the default value. This value is used as the second argument in calls to `isinstance`, so it has to follow that function's requirements (class, type or a tuple containing classes, types or tuples).
- **op** – An optional operation that is applied to the underlying value when `get()` is called.

- **required** – If True, the *get()* method will raise an error if the reference does not contain a value. This argument has no effect when a default value is provided. Setting this to True will raise an error if *op* is not None.

Raises

- `TypeError` – If *field_type* is not None and is different from the type of the default value.
- `ValueError` – If both the default value and *field_type* is None.

`empty()`

Returns True if the reference points to a None value.

`get()`

Gets the value of the *FieldReference* object.

This will dereference *_pointer* and apply all ops to its value.

Returns The result of applying all ops to the dereferenced pointer.

Raises `RequiredValueError` – if *required* is True and the underlying value for the reference is False.

`has_cycle(visited=None)`

Finds cycles in the reference graph.

Parameters **visited** – Set containing the ids of all visited nodes in the graph. The default value is the empty set.

Returns True if there is a cycle in the reference graph.

`set(value, type_safe=True)`

Overwrites the value pointed by a *FieldReference*.

Parameters

- **value** – New value.
- **type_safe** – Check that old and new values are of the same type.

Raises

- `TypeError` – If *type_safe* is true and old and new values are not of the same type.
- `MutabilityError` – If a cycle is found in the reference graph.

3.4 Additional Methods

<code>create(**kwargs)</code>	Creates a <i>ConfigDict</i> with the given named arguments as key-value pairs.
<code>placeholder(field_type[, required])</code>	Defines an entry in a <i>ConfigDict</i> that has no value yet.
<code>required_placeholder(field_type)</code>	Defines an entry in a <i>ConfigDict</i> with unknown but required value.
<code>recursive_rename(conf, old_name, new_name)</code>	Returns copy of <i>conf</i> with <i>old_name</i> recursively replaced by <i>new_name</i> .
<code>CustomJSONEncoder(*[, skipkeys, ...])</code>	JSON encoder for <i>ConfigDict</i> and <i>FieldReference</i> .
<code>JSONDecodeError</code>	
<code>MutabilityError</code>	
<code>RequiredValueError</code>	

3.4.1 ml_collections.config_dict.create

`ml_collections.config_dict.create(**kwargs)`

Creates a *ConfigDict* with the given named arguments as key-value pairs.

This allows for simple dictionaries whose elements can be accessed directly using field access:

```
from ml_collections import config_dict
point = config_dict.create(x=1, y=2)
print(point.x, point.y)
```

This is particularly useful for compactly writing nested configurations:

```
config = config_dict.create(
    data=config_dict.create(
        game='freeway',
        frame_size=100),
    model=config_dict.create(num_hidden=1000))
```

The reason for the existence of this function is that it simplifies the code required for the majority of the use cases of *ConfigDict*, compared to using either *ConfigDict* or *namedtuple*'s. Examples of such use cases include training script configuration, and returning multiple named values.

Parameters ****kwargs** – key-value pairs to be stored in the *ConfigDict*.

Returns A *ConfigDict* containing the key-value pairs in *kwargs*.

3.4.2 ml_collections.config_dict.placeholder

`ml_collections.config_dict.placeholder(field_type, required=False)`

Defines an entry in a *ConfigDict* that has no value yet.

Example:

```
config = configdict.create(
    batch_size = configdict.placeholder(int),
    frame_shape = configdict.placeholder(tf.TensorShape))
```

Parameters

- **field_type** – type of value.
- **required** – If True, the placeholder will raise an error on access if the underlying value hasn't been set.

Returns A *FieldReference* with value None and the given type.

3.4.3 ml_collections.config_dict.required_placeholder

`ml_collections.config_dict.required_placeholder(field_type)`

Defines an entry in a *ConfigDict* with unknown but required value.

Example:

```

config = configdict.create(
    batch_size = configdict.required_placeholder(int))

try:
    print(config.batch_size)
except RequiredValueError:
    pass

config.batch_size = 10
print(config.batch_size) # 10

```

Parameters `field_type` – type of value.

Returns A *FieldReference* with value None and the given type.

3.4.4 ml_collections.config_dict.recursive_rename

`ml_collections.config_dict.recursive_rename` (*conf*, *old_name*, *new_name*)

Returns copy of *conf* with *old_name* recursively replaced by *new_name*.

This is not done in place, no changes are made to *conf* but a new *ConfigDict* is returned with the changes made. This is useful if the name of a parameter has been changed in code but you need to load an old config.

Example usage: `updated_conf = configdict.recursive_rename(conf, "config", "kwargs")`

Parameters

- **conf** – a *ConfigDict* which needs updating
- **old_name** – the name used in the *ConfigDict* which is out of sync with the code
- **new_name** – the name used in the code

Returns A *ConfigDict* which is a copy of *conf* but with all instances of *old_name* replaced with *new_name*.

3.4.5 ml_collections.config_dict.CustomJSONEncoder

```

class ml_collections.config_dict.CustomJSONEncoder(*,
                                                    skipkeys=False,
                                                    ensure_ascii=True,
                                                    check_circular=True,
                                                    allow_nan=True,
                                                    sort_keys=False,
                                                    indent=None,
                                                    separators=None,
                                                    default=None)

```

JSON encoder for *ConfigDict* and *FieldReference*.

The encoder throws an exception for non-supported types.

```

__init__(*, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True,
         sort_keys=False, indent=None, separators=None, default=None)

```

Constructor for *JSONEncoder*, with sensible defaults.

If *skipkeys* is false, then it is a *TypeError* to attempt encoding of keys that are not *str*, *int*, *float* or *None*. If *skipkeys* is *True*, such items are simply skipped.

If *ensure_ascii* is true, the output is guaranteed to be *str* objects with all incoming non-ASCII characters escaped. If *ensure_ascii* is false, the output can contain non-ASCII characters.

If `check_circular` is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an `OverflowError`). Otherwise, no such check takes place.

If `allow_nan` is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a `ValueError` to encode such floats.

If `sort_keys` is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If `indent` is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if `indent` is None and (',', ':') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

Methods

<code>__init__</code> (*[, skipkeys, ensure_ascii, ...])	Constructor for <code>JSONEncoder</code> , with sensible defaults.
<code>default</code> (obj)	Implement this method in a subclass such that it returns a serializable object for <code>o</code> , or calls the base implementation (to raise a <code>TypeError</code>).
<code>encode</code> (o)	Return a JSON string representation of a Python data structure.
<code>iterencode</code> (o[, _one_shot])	Encode the given object and yield each string representation as available.

Attributes

<code>item_separator</code>
<code>key_separator</code>

3.4.6 ml_collections.config_dict.JSONDecodeError

exception `ml_collections.config_dict.JSONDecodeError`

3.4.7 ml_collections.config_dict.MutabilityError

exception `ml_collections.config_dict.MutabilityError`

3.4.8 ml_collections.config_dict.RequiredValueError

exception `ml_collections.config_dict.RequiredValueError`

ml_collections.config_flags package

Config flags module.

4.1 DEFINE_config_dict

```
config_flags.DEFINE_config_dict (config:      ml_collections.config_dict.config_dict.ConfigDict,
                                help_string: str      = 'ConfigDict instance.',
                                flag_values:  absl.flags._flagvalues.FlagValues  =
                                <absl.flags._flagvalues.FlagValues object>, lock_config:
                                bool = True, sys_argv: Optional[List[str]] = None, **kwargs)
                                → absl.flags._flagvalues.FlagHolder
```

Defines flag for inline *ConfigDict*'s compatible with absl flags.

Similar to *DEFINE_config_file* except the flag's value should be a *ConfigDict* instead of a path to a file containing a *ConfigDict*. After the flag is parsed, *FLAGS.name* will contain a reference to the *ConfigDict*, optionally with some values overridden.

Typical usage example:

script.py:

```
from ml_collections import config_dict
from ml_collections import config_flags

config = config_dict.ConfigDict({
    'field1': 1,
    'field2': 'tom',
    'nested': {
        'field': 2.23,
    }
})
```

(continues on next page)

(continued from previous page)

```

_CONFIG = config_flags.DEFINE_config_dict('my_config', config)
...

print(_CONFIG.value)

```

The following command:

```

python script.py -- --my_config.field1 8
                  --my_config.nested.field=2.1

```

will print:

```

field1: 8
field2: tom
nested: {field: 2.1}

```

Parameters

- **name** – Flag name.
- **config** – *ConfigDict* object.
- **help_string** – Help string to display when `-helpfull` is called. (default: “ConfigDict instance.”)
- **flag_values** – *FlagValues* instance used for parsing. (default: `absl.flags.FLAGS`)
- **lock_config** – If set to `True`, loaded config will be locked through calling `.lock()` method on its instance (if it exists). (default: `True`)
- **sys_argv** – If set, interprets this as the full list of args used in parsing. This is used to identify which overrides to define as flags. If not specified, uses the system `sys.argv` to figure it out.
- ****kwargs** – Optional keyword arguments passed to *Flag* constructor.

Returns a handle to defined flag.

4.2 DEFINE_config_file

`config_flags.DEFINE_config_file` (default: *Optional[str] = None*, *help_string: str = 'path to config file.'*, *flag_values: absl.flags._flagvalues.FlagValues = <absl.flags._flagvalues.FlagValues object>*, *lock_config: bool = True*, *sys_argv: Optional[List[str]] = None*, ***kwargs*) → `absl.flags._flagvalues.FlagHolder`

Defines flag for *ConfigDict* files compatible with *absl* flags.

The flag’s value should be a path to a valid python file which contains a function called `get_config()` that returns a python object specifying a configuration. After the flag is parsed, `FLAGS.name` will contain a reference to this object, optionally with some values overridden.

During flags parsing, every flag of form `-name.([a-zA-Z0-9]+.?)+=value` and `-name.([a-zA-Z0-9]+.?) value` will be treated as an override of a specific field in the config object returned by this flag. Field is essentially a dot delimited path inside the object where each path element has to be either an attribute or a key existing in the config object. For example `-my_config.field1.field2=val` means “assign value `val` to the attribute (or key) `field2` inside value of the attribute (or key) `field1` inside the value of `my_config` object”. If there are both attribute and key-based access with the same name, attribute is preferred.

Typical usage example:

script.py:

```
from ml_collections import config_flags

_CONFIG = config_flags.DEFINE_config_file('my_config')

print(_CONFIG.value)
```

config.py:

```
def get_config():
    return {
        'field1': 1,
        'field2': 'tom',
        'nested': {
            'field': 2.23,
        },
    }
```

The following command:

```
python script.py -- --my_config=config.py
                  --my_config.field1 8
                  --my_config.nested.field=2.1
```

will print:

```
{'field1': 8, 'field2': 'tom', 'nested': {'field': 2.1}}
```

It is possible to parameterise the `get_config` function, allowing it to return a differently structured result for different occasions. This is particularly useful when setting up hyperparameter sweeps across various network architectures.

parameterised_config.py:

```
def get_config(config_string):
    possible_configs = {
        'mlp': {
            'constructor': 'snt.nets.MLP',
            'config': {
                'output_sizes': (128, 128, 1),
            }
        },
        'lstm': {
            'constructor': 'snt.LSTM',
            'config': {
                'hidden_size': 128,
                'forget_bias': 1.0,
            }
        }
    }
    return possible_configs[config_string]
```

If a colon is present in the command line override for the config file, everything to the right of the colon is passed into the `get_config` function. The following command lines will both function correctly:

```
python script.py -- --my_config=parameterised_config.py:mlp
                    --my_config.config.output_sizes="(256,256,1) "

python script.py -- --my_config=parameterised_config.py:lstm
                    --my_config.config.hidden_size=256
```

The following will produce an error, as the `hidden_size` flag does not exist when the “mlp” `config_string` is provided:

```
python script.py -- --my_config=parameterised_config.py:mlp
                    --my_config.config.hidden_size=256
```

Parameters

- **name** – Flag name, optionally including extra config after a colon.
- **default** – Default value of the flag (default: None).
- **help_string** – Help string to display when `-helpfull` is called. (default: “path to config file.”)
- **flag_values** – FlagValues instance used for parsing. (default: `absl.flags.FLAGS`)
- **lock_config** – If set to True, loaded config will be locked through calling `.lock()` method on its instance (if it exists). (default: True)
- **sys_argv** – If set, interprets this as the full list of args used in parsing. This is used to identify which overrides to define as flags. If not specified, uses the system `sys.argv` to figure it out.
- ****kwargs** – Optional keyword arguments passed to Flag constructor.

Returns a handle to defined flag.

4.3 Additional Methods

<code>ml_collections.config_flags.config_flags.is_config_flag(flag)</code>	Returns True iff <i>flag</i> is an instance of <code>_ConfigFlag</code> .
<code>ml_collections.config_flags.config_flags.GetValue(...)</code>	Gets value of a single field.
<code>ml_collections.config_flags.config_flags.GetType(...)</code>	Gets type of field in config described by a <code>config_path</code> .
<code>ml_collections.config_flags.config_flags.GetTypes</code>	
<code>ml_collections.config_flags.config_flags.SetValue(...)</code>	Sets value of field described by <code>config_path</code> .

4.3.1 ml_collections.config_flags.config_flags.is_config_flag

`ml_collections.config_flags.config_flags.is_config_flag(flag)`
Returns True iff *flag* is an instance of `_ConfigFlag`.

External users of the library may need to check if a flag is of this type or not, particularly because `ConfigFlags` should be parsed before any other flags. This function allows that test to be done without making the whole

class public.

Parameters `flag` – Flag object.

Returns True iff `isinstance(flag, _ConfigFlag)` is true.

4.3.2 ml_collections.config_flags.config_flags.GetValue

`ml_collections.config_flags.config_flags.GetValue` (`config_path: str, config: Any`)
Gets value of a single field.

Example usage:

```
>>> config = {'a': {'b', {'c', 10}}}
>>> assert config_path.get_value('a.b.c', config) == 10
```

Parameters

- `config_path` – Any string that `split` can process.
- `config` – A nested datastructure

Returns The last object when walking config with `config_path`.

Raises

- `IndexError` – Integer field not found in nested structure.
- `KeyError` – Non-integer field not found in nested structure.
- `ValueError` – Empty/invalid `config_path` after parsing.

4.3.3 ml_collections.config_flags.config_flags.GetType

`ml_collections.config_flags.config_flags.GetType` (`config_path: str, config: Any, normalize=True`)
Gets type of field in config described by a `config_path`.

Example usage:

```
>>> config = {'a': {'b', {'c', 10}}}
>>> assert config_path.get_type('a.b.c', config) is int
```

Parameters

- `config_path` – Any string that `split` can process.
- `config` – A nested datastructure
- `normalize` – whether to normalize the type (in particular
- **Optional annotations on dataclass fields) (`strip`)** –

Returns The type of last object when walking config with `config_path`.

Raises

- `IndexError` – Integer field not found in nested structure.
- `KeyError` – Non-integer field not found in nested structure.
- `ValueError` – Empty/invalid `config_path` after parsing.

- `TypeError` – Ambiguous type annotation on dataclass field.

4.3.4 ml_collections.config_flags.config_flags.SetValue

`ml_collections.config_flags.config_flags.SetValue` (*config_path: str, config: Any, value: Any*)

Sets value of field described by `config_path`.

Example usage:

```
>>> config = {'a': {'b', {'c', 10}}}  
>>> config_path.set_value('a.b.c', config, 20)  
>>> assert config['a']['b']['c'] == 20
```

Parameters

- **config_path** – Any string that *split* can process.
- **config** – A nested datastructure
- **value** – A value to assign to final field.

Raises

- `IndexError` – Integer field not found in nested structure.
- `KeyError` – Non-integer field not found in nested structure.
- `ValueError` – Empty/invalid `config_path` after parsing.

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

5.1 Contributor License Agreement

Contributions to this project must be accompanied by a Contributor License Agreement (CLA). You (or your employer) retain the copyright to your contribution; this simply gives us permission to use and redistribute your contributions as part of the project. Head over to <https://cla.developers.google.com/> to see your current agreements on file or to sign a new one.

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

5.2 Code reviews

All submissions, including submissions by project members, require review. We use GitHub pull requests for this purpose. Consult [GitHub Help](#) for more information on using pull requests.

5.3 Community Guidelines

This project follows [Google's Open Source Community Guidelines](#).

CHAPTER 6

Testing

TODO(mohitreddy): Add sections to install bazel and run tests.

m

`ml_collections.config_dict`, 7
`ml_collections.config_flags`, 19

Symbols

- `__init__()` (*ml_collections.config_dict.ConfigDict* method), 8
- `__init__()` (*ml_collections.config_dict.CustomJSONEncoder* method), 17
- `__init__()` (*ml_collections.config_dict.FieldReference* method), 14
- `__init__()` (*ml_collections.config_dict.FrozenConfigDict* method), 13
- ## C
- `ConfigDict` (class in *ml_collections.config_dict*), 7
- `convert_dict()` (*ml_collections.config_dict.ConfigDict* attribute), 8
- `copy_and_resolve_references()` (*ml_collections.config_dict.ConfigDict* method), 8
- `create()` (in module *ml_collections.config_dict*), 16
- `CustomJSONEncoder` (class in *ml_collections.config_dict*), 17
- ## D
- `DEFINE_config_dict()` (*ml_collections.config_flags* method), 19
- `DEFINE_config_file()` (*ml_collections.config_flags* method), 20
- ## E
- `empty()` (*ml_collections.config_dict.FieldReference* method), 15
- `eq_as_configdict()` (*ml_collections.config_dict.ConfigDict* method), 9
- ## F
- `FieldReference` (class in *ml_collections.config_dict*), 14
- `FrozenConfigDict` (class in *ml_collections.config_dict*), 13
- ## G
- `get()` (*ml_collections.config_dict.ConfigDict* method), 9
- `get()` (*ml_collections.config_dict.FieldReference* method), 15
- `get_oneway_ref()` (*ml_collections.config_dict.ConfigDict* method), 9
- `get_type()` (*ml_collections.config_dict.ConfigDict* method), 9
- `GetType()` (in module *ml_collections.config_flags.config_flags*), 23
- `GetValue()` (in module *ml_collections.config_flags.config_flags*), 23
- ## H
- `has_cycle()` (*ml_collections.config_dict.FieldReference* method), 15
- ## I
- `ignore_type()` (*ml_collections.config_dict.ConfigDict* method), 9
- `is_config_flag()` (in module *ml_collections.config_flags.config_flags*), 22
- `is_locked()` (*ml_collections.config_dict.ConfigDict* attribute), 9
- `is_type_safe()` (*ml_collections.config_dict.ConfigDict* attribute), 9
- `items()` (*ml_collections.config_dict.ConfigDict* method), 9
- `iteritems()` (*ml_collections.config_dict.ConfigDict* method), 10
- `iterkeys()` (*ml_collections.config_dict.ConfigDict* method), 10
- `itervalues()` (*ml_collections.config_dict.ConfigDict* method), 10

J

JSONDecodeError, 18

Kkeys() (ml_collections.config_dict.ConfigDict
method), 10**L**lock() (ml_collections.config_dict.ConfigDict
method), 10**M**

ml_collections.config_dict (module), 7

ml_collections.config_flags (module), 19

MutabilityError, 18

Pplaceholder() (in module
ml_collections.config_dict), 16**R**recursive_rename() (in module
ml_collections.config_dict), 17required_placeholder() (in module
ml_collections.config_dict), 16

RequiredValueError, 18

Sset() (ml_collections.config_dict.FieldReference
method), 15SetValue() (in module
ml_collections.config_flags.config_flags),
24**T**to_dict() (ml_collections.config_dict.ConfigDict
method), 10to_json() (ml_collections.config_dict.ConfigDict
method), 10to_json_best_effort()
(ml_collections.config_dict.ConfigDict
method), 11to_yaml() (ml_collections.config_dict.ConfigDict
method), 11**U**unlock() (ml_collections.config_dict.ConfigDict
method), 11unlocked() (ml_collections.config_dict.ConfigDict
method), 11update() (ml_collections.config_dict.ConfigDict
method), 11update_from_flattened_dict()
(ml_collections.config_dict.ConfigDict
method), 12**V**values() (ml_collections.config_dict.ConfigDict
method), 13