## CONTENTS

1 Schema types ..................................................  1

2 Exceptions ..........................................................  5

3 How – yamap – YAML to Python! ..........................  7
   3.1 Basic example ...................................................  7
   3.2 Mapping YAML types to Python ...............................  8
   3.3 Type conversion ..................................................  9

Python Module Index ..............................................  11

Index ......................................................................  13
class yamap.schema.yatype
Concrete schema type that represents one or more parts of the YAML file.

class yamap.schema.yanode
Abstract class that represents one node of the YAML parse tree. Implements a generic matching algorithm using
the nodes YAML tag.

__init__(tag: str, type: Callable)

Parameters

• tag – Regular expression string to match YAML node tag against.
• type – Output type conversion function. Allows converting basic construction results
into more complex types.

load(stream: Any) → Any
Helper method that tries matching the given stream to self.

matches(node: Node) → yatype
Will try to match the given YAML node to a concrete yatype using fullmatch(self.tag, node.
tag).

class yamap.schema.yascalar
Helper class representing all kinds of YAML scalar nodes. Can be used as a catch-all.

__init__(tag: str, type: Callable, construct: Callable, value: str)

Parameters

• tag – See yanode.
• type – See yanode.
• value – Regular expression string to match YAML node value against.
• construct – Basic YAML to Python construction function.

construct_leaf(SafeConstructor, Node) → Any
Turns the YAML node into a Python object by calling this objects construct member.

matches(node: Node) → yatype
Will try to match the given YAML node to a concrete yatype using fullmatch(self.tag, node.
tag) and if the value member variable is set also fullmatch(self.value, node.value).

class yamap.schema.yaint
More specific scalar type to only match the YAML ints.

class yamap.schema.yafloat
More specific scalar type to only match the YAML floats.
**class yamap.schema.yanumber**  
More specific scalar type to only match the YAML numbers.

**class yamap.schema.yastr**  
More specific scalar type to only match string.

**class yamap.schema.yabool**  
More specific scalar type to only match the YAML booleans.

**class yamap.schema.yanull**  
More specific scalar type to only match the YAML null value.

**class yamap.schema.yaseq**  
Represents a YAML sequence node.

```python
__init__(tag: str = 'tag:yaml.org,2002:seq', type: Callable = list, unpack: bool = False)
```

**Parameters**

- **tag** – See `yanode`.
- **type** – See `yanode`.
- **unpack** – If `True` call `type` with the nodes children as unpacked positional arguments.

**case**(schema: yatype) → yaseq  
Register an additional schema type option.

**class yamap.schema.yamap**  
Represents a YAML mapping node.

```python
__init__(tag: str = 'tag:yaml.org,2002:map', type: Callable = list, squash: bool = False, unpack: bool = False)
```

**Parameters**

- **tag** – See `yanode`.
- **type** – See `yanode`.
- **unpack** – If `True` call `type` with the nodes children as unpacked keyword arguments.
- **unpack** – If `True` call `type` only with the nodes first child as argument.

**zero_or_one**(regex: str, schema: yatype, type: Callable = pair) → yamap  
Shortcut to add an optional yaentry to this mapping.

**exactly_one**(regex: str, schema: yatype, type: Callable = pair) → yamap  
Shortcut to add a required yaentry to this mapping.

**zero_or_more**(regex: str, schema: yatype, type: Callable = pair) → yamap  
Shortcut to add an optional and repeatable yaentry to this mapping.

**one_or_more**(regex: str, schema: yatype, type: Callable = pair) → yamap  
Shortcut to add a required and repeatable yaentry to this mapping.

**case**(regex: str, schema: yatype, type: Callable = pair, repeat: bool = False, required: bool = False) → yamap  
Shortcut to add a yaentry to this mapping without having to go through object creation.

**entry**(entry: yaentry) → yamap  
Add a yaentry to this mapping.

**class yamap.schema.yaentry**  
Possible schema type of an mapping entry. Supports various options to configure how and what key-value pairs are matched.
__init__ (required: bool = False, repeat: bool = False)

Parameters

- **required** – If True this entry has to appear at least once in its associated `yamap`.
- **repeat** – If True this entry can appear multiple times in its associated `yamap`.

case (pattern: str, schema: yatype, type: Callable = pair) → yaentry

Add a possible key-type pair to the alternatives matched by this entry.

class yamap.schema.yaoneof

Virtual schema class that represents the an either-or of types possible at this point in the hierarchy.

__init__ (*entries: yatype)

Parameters entries – Possible schema types this node will match.

case (entry: yatype) → yaoneof

Register an additonal schema type option.
**CHAPTER TWO**

**EXCEPTIONS**

- **except**ion `yamap.errors.MappingError(msg, node)`
  
  Exception for any kind or error that occurs while mapping YAML to Python objects using a yamap schema.

- **except**ion `yamap.errors.NoMatchingType(node)`
  
  Specific MappingError that occurs when yamap can find no schema type to handle a YAML node.
Using complex YAML configuration files in Python used to be painful. Until now you had the choice of

- reading into a datastructure of basic types (dicts, lists, scalars) and then littering your code with traversal logic and dictionary keys,
- spreading a big helping of custom YAML tags over your files and adding constructors to turn these into data classes,
- or wrestling with PyYAML’s experimental path constructors.

None of these is fun and only the last one halfhartedly solves to issue of validating the structure of your files.

**yamap** to the rescue. It’s the bastard child of a YAML to Python data mapper and a schema definition DSL and validator.

### 3.1 Basic example

Given a YAML file

```yaml
command: /usr/bin/echo
arguments:
  - Hello
  - world,
  - ${NAME}
env: { NAME: Bob }
capture: true
timeout: 5.5
```

you can easily us yamap to parse that

```python
from yamap import *

schema = (  
yamap()  
  .exactly_one('command', yastr(value='/.+'))  
  .zero_or_one('arguments', yaseq().case(yastr))  
  .zero_or_one('env', yamap().zero_or_more('[A-Z]+', yastr))  
  .zero_or_one('capture', yabool)  
  .zero_or_one('timeout', yanumber)
)

result = schema.load(...)  
```

which will return you this Python object tree
Easy but at first glance no different than using a plain YAML parser like pyyaml or ruamel.yaml so let’s dive a bit deeper what else yamap has in store for you.

### 3.2 Mapping YAML types to Python

To use yamap you’ll need to define a schema tree of data types that describe the layout of the YAML data you are expecting to parse. Loading a file that doesn’t adhere to this schema will make yamap throw a MappingError. All schema types are immutable but provide methods that return modified copies of them. This allows easy method chaining and reuse of schema trees within or across schemas.

Schema types are matched to the YAML data by means of the YAML tags returned by the YAML parser.

#### 3.2.1 Scalar data types

These are the leafs of a YAML schema tree and get turned into the corresponding simple Python objects:

- `yaint` matches tag:yaml.org,2002:int and constructs Python `int`
- `yafloat` matches tag:yaml.org,2002:float and constructs Python `float`
- `yanumber` matches any of the two aforementioned types and constructs either Python `int` or `float` accordingly.
- `yastr` matches tag:yaml.org,2002:str and constructs Python `str`
- `yabool` matches tag:yaml.org,2002:bool and constructs Python `bool`
- `yanull` matches tag:yaml.org,2002:null and constructs Python `None`
- `yascal` matches any of the aforementioned types and constructs accordingly.

In addition to matching by tag, you can pass yascal (and thus all other scalar types as these derive from it) a regex on construction. yamap will try to fullmatch() this against the node value (being the unconstructed str from the YAML data) and throw a MappingError if this doesn’t succeed.

#### 3.2.2 Sequence type

The schema type `yaseq` matches tag:yaml.org,2002:seq will construct it’s children and return them as a Python list.

As such you’ll need to specify what types are allowed as members of the list using case()

These are the branches of the YAML schema tree. The will be evaluated once before and once after their children. First to construct a mapping of children to schema types and second to actuall turn them into Python types:

- `yaseq` matches tag:yaml.org,2002:seq and constructs Python `list`
- `yamap` matches tag:yaml.org,2002:map and constructs Python `dict`
3.2.3 Virtual data types

These don’t map to a YAML node directly but act as helpers to express more complex schema hierarchies:

- *yaoneof* provides a switch-case like alternation between schema types.
- *yaentry* helps in matching key-value pairs of a *yamap*.

3.3 Type conversion

All none-virtual schema types support passing a callable as the constructor argument *type*. That callable will be passed the constructed value as its single argument. For *yaseq* types that is a list of elements and for *yamap* a list of key,value tuples. It is unrestricted in its return type.

3.3.1 Unpacking

Sometimes you’d rather have your *type* be called with unpacked arguments or keyword arguments instead of a single list. That’s what happens when you set *unpack=True* when creating a *yaseq* or *yamap* respectively.

```python
def hello(what='world', who='friend'):
    return f'Hello {what}, my {who}!'.format(what=what, who=who)

schema = (yamap(type=hello, unpack=True)
    .zero_or_one('what', yastr)
    .zero_or_one('who', yastr)
)
assert schema.load('{ who: "mate" }') == 'Hello world, my mate!'```

3.3.2 Map squashing

Setting *squash=True* when creating a *yamap* will make the type converter be called with only the first key,value pair of your mapping.

For example instead of having to write

```
- name: upper
- name: replace
  args: [FRIEND, MATE]
- name: replace
  args: [HELLO, HEY]
```

you can implement a schema to parse the much more readable

```
- upper
  replace: [FRIEND, MATE]
  replace: [HELLO, HEY]
```

like this
from yamap import *
from dataclasses import dataclass

def pipeline(items):
    input = 'Hello world, my friend!'  
    for item in items:
        input = getattr(input, item.name)(*item.args)
    return input

@dataclass
class entry:
    name: str
    args: tuple = ()

schema = (
    yaseq(type=pipeline)
    .case(yastr(type=entry))
    .case(yamap(type=entry, squash=True, unpack=True)
        .exactly_one('.+', yaseq().case(yascalar)))
)
PYTHON MODULE INDEX

y
yamap, ??
yamap
Symbols

__init__() (yamap.schema.yaentry method), 2
__init__() (yamap.schema.yamap method), 2
__init__() (yamap.schema.yanode method), 1
__init__() (yamap.schema.yaoneof method), 3
__init__() (yamap.schema.yascalar method), 1
__init__() (yamap.schema.yaseq method), 2

C

case() (yamap.schema.yaentry method), 3
case() (yamap.schema.yamap method), 2
case() (yamap.schema.yaoneof method), 3
case() (yamap.schema.yaseq method), 2
construct_leaf() (yamap.schema.yascalar method), 1

E

time() (yamap.schema.yamap method), 2
extactly one() (yamap.schema.yamap method), 2

L

load() (yamap.schema.yanode method), 1

M

MappingError, 5
matches() (yamap.schema.yanode method), 1
matches() (yamap.schema.yascalar method), 1
module
    yamap, 1

N

NoMatchingType, 5

O

one_or_more() (yamap.schema.yamap method), 2

Y

yabool (class in yamap.schema), 2
yaentry (class in yamap.schema), 2
yafloat (class in yamap.schema), 1
yaint (class in yamap.schema), 1

yamap
    module, 1
yamap (class in yamap.schema), 2
yanode (class in yamap.schema), 1
yanull (class in yamap.schema), 2
yanumber (class in yamap.schema), 1
yaoe (class in yamap.schema), 3
yascalar (class in yamap.schema), 1
yaseq (class in yamap.schema), 2
yast (class in yamap.schema), 2
yatpe (class in yamap.schema), 1

Z

zero_or_more() (yamap.schema.yamap method), 2
zero_or_one() (yamap.schema.yamap method), 2