sanest

Release

Jun 27, 2017

Contents

sane nested dictionaries and lists

Sample JSON input:

```
{
  "data": {
    "users": [
      {"id": 12, "name": "alice"},
      {"id": 34, "name": "bob"}
    ]
  }
}
```

Without `sanest`:

```
d = json.loads(...)
for user in d['data']['users']:
    print(user['name'])
```

With `sanest`:

```
d = json.loads(...)
wrapped = sanest.dict.wrap(d)
for user in wrapped['data', 'users':[dict]]:
    print(user['name':str])
```

The code is now type-safe and will fail fast on unexpected input data.

Table of contents

CHAPTER 1

Overview

`sanest` is a Python library that makes it easy to consume, produce, or modify nested JSON structures in a strict and type-safe way. It provides two container data structures, specifically designed for the JSON data model:

- `sanest.dict`
- `sanest.list`

These are thin wrappers around the built-in `dict` and `list`, with minimal overhead and an almost identical API, but with a few new features that the built-in containers do not have:

- nested operations
- type checking
- data model restrictions

These features are very easy to use: with minimal code additions, otherwise implicit assumptions about the nesting structure and the data types can be made explicit, adding type-safety and robustness.

`sanest` is *not* a validation library. It aims for the sweet spot between 'let's hope everything goes well' (if not, unexpected crashes or undetected buggy behaviour ensues) and rigorous schema validation (lots of work, much more code).

In practice, `sanest` is especially useful when crafting requests for and processing responses from third-party JSON-based APIs, but is by no means limited to this use case.

CHAPTER 2

Installation

Use `pip` to install `sanest` into a `virtualenv`:

```
pip install sanest
```

`sanest` requires Python 3.3+ and has no additional dependencies.

CHAPTER 3

Why `sanest`?

Consider this JSON data structure, which is a stripped-down version of the example JSON response from the GitHub issues API documentation:

```
{
  "id": 1,
  "state": "open",
  "title": "Found a bug",
  "user": {
    "login": "octocat",
    "id": 1,
  },
  "labels": [
    {
      "id": 208045946,
      "name": "bug"
    }
  ],
  "milestone": {
    "id": 1002604,
    "state": "open",
    "title": "v1.0",
    "creator": {
      "login": "octocat",
      "id": 1,
    },
    "open_issues": 4,
    "closed_issues": 8,
  }
}
```

The following code prints all labels assigned to this issue, using only lowercase letters:

```
>>> issue = json.load(...)
>>> for label in issue['labels']:
```

```
...     print(label['name'].lower())
bug
```

Hidden asssumptions. This code does work for valid inputs, but makes quite a few implicit assumptions about that input:

- The result of `json.load()` is a dictionary.
- The `labels` field exists.
- The `labels` field points to a list.
- This list contains zero or more dictionaries.
- These dictionaries have a `name` field.
- The `name` field points to a string.

When presented with input data fow which these assumptions do not hold, various things can happen. For instance:

- Accessing `d['labels']` raises `KeyError` when the field is missing.
- Accessing `d['labels']` raises `TypeError` if it is not a `dict`.

  The actual exception messages vary and can be confusing:

  - `TypeError: string indices must be integers`
  - `TypeError: list indices must be integers or slices, not str`
  - `TypeError: 'NoneType' object is not subscriptable`

- If the `labels` field is not a list, the `for` loop may raise a `TypeError`, but not in all cases.

  If `labels` contained a string or a dictionary, the `for` loop will succeed, since strings and dictionaries are iterable, and loop over the individual characters of this string or over the keys of the dictionary. This was not intended, but will not raise an exception.

  In this example, the next line will crash, since the `label['name']` lookup will fail with a `TypeError` telling that `string indices must be integers`, but depending on the code everything may seem fine even though it really is not.

The above is not an exhaustive list of things that can go wrong with this code, but it gives a pretty good overview.

Validation. One approach of safe-guarding against the issues outlined above would be to write validation code. There are many validation libraries, such as jsonschema, Marshmallow, Colander, Django REST framework, and many others, that are perfectly suitable for this task.

The downside is that writing the required schema definitions is a lot of work. A strict validation step will also make the code much larger and hence more complex. Especially when dealing with data formats that are not 'owned' by the application, e.g. when interacting with a third-party REST API, this may be a prohibitive amount of effort.

In the end, rather than going through all this extra effort, it may be simpler to just use the code above as-is and hope for the best.

The sane approach. However, there are more options than full schema validation and no validation at all. This is what `sanest` aims for: a sane safety net, without going overboard with upfront validation.

Here is the equivalent code using `sanest`:

```
>>> issue = sanest.dict.wrap(json.loads(...))    # 1
>>> for user in issue['labels':[dict]]:          # 2
...     print(label['name':str].lower())         # 3
bug
```

While the usage of slice syntax for dictionary lookups and using the built-in types directly (e.g. `str` and `dict`) may look a little surprising at first, the code is actually very readable and explicit.

Here is what it does:

1. Create a thin `dict` wrapper.

   This ensures that the input is a dictionary, and enables the type checking lookups used in the following lines of code.

2. Look up the `labels` field.

   This ensures that the field contains a list of dictionaries. 'List of dictionaries' is condensely expressed as `[dict]`, and passed to the `d[...]` lookup using slice syntax (with a colon).

3. Print the lowercase value of the `name` field.

   This checks that the value is a string before calling `.lower()` on it.

This code still raises `KeyError` for missing fields, but any failed check will immediately raise a very clear exception with a meaningful message detailing what went wrong.

CHAPTER 4

Data model

The JSON data model is restricted, and `sanest` strictly adheres to it. `sanest` uses very strict type checks and will reject any values not conforming to this data model.

Containers. There are two container types, which can have arbitrary nesting to build more complex structures:

- `sanest.dict` is an unordered collection of named items.
- `sanest.list` is an ordered collection of values.

In a dictionary, each item is a `(key, value)` pair, in which the key is a unique string (`str`). In a list, values have an associated index, which is an integer counting from zero.

Leaf values. Leaf values are restricted to:

- strings (`str`)
- integer numbers (`int`)
- floating point numbers (`float`)
- booleans (`bool`)
- `None` (no value, encoded as `null` in JSON)

CHAPTER 5

Basic usage

`sanest` provides two classes, `sanest.dict` and `sanest.list`, that behave very much like the built-in `dict` and `list`, supporting all the regular operations such as getting, setting, and deleting items.

To get started, import the `sanest` module:

```
import sanest
```

Dictionary. The `sanest.dict` constructor behaves like the built-in `dict` constructor:

```
d = sanest.dict(regular_dict_or_mapping)
d = sanest.dict(iterable_with_key_value_pairs)
d = sanest.dict(a=1, b=2)
```

Usage examples (see API docs for details):

```
d = sanest.dict(a=1, b=2)
d['a']
d['c'] = 3
d.update(d=4)
d.get('e', 5)
d.pop('f', 6)
del d['a']
for v in d.values():
    print(v)
d.clear()
```

List. The `sanest.list` constructor behaves like the built-in `list` constructor:

```
l = sanest.list(regular_list_or_sequence)
l = sanest.list(iterable)
```

Usage examples (see API docs for details):

```
l = sanest.list([1, 2])
l[0]
```

```
l.append(3)
l.extend([4, 5])
del l[0]
for v in l():
    print(v)
l.pop()
l.count(2)
l.sort()
l.clear()
```

Container values. Operations that return a nested dictionary or list will always be returned as a `sanest.dict` or `sanest.list`:

```
>>> issue['user']
sanest.dict({"login": "octocat", "id": 1})
```

Operations that accept a container value from the application, will accept regular `dict` and `list` instances, as well as `sanest.dict` and `sanest.list` instances:

```
>>> normal_dict = {'a': 1, 'b': 2}
>>> issue['x'] = normal_dict
>>> other = sanest.dict()
>>> other['a'] = 1
>>> issue['y'] = other
```

CHAPTER 6

Nested operations

In addition to normal dictionary keys (`str`) and list indices (`int`), `sanest.dict` and `sanest.list` can operate directly on values in a nested structure. Nested operations work like normal container operations, but instead of a single key or index, they use a path that points into nested dictionaries and lists.

Path syntax. A path is simply a sequence of strings (dictionary keys) and integers (list indices). Here are some examples for the Github issue JSON example from a previous section:

```
'user', 'login'
'labels', 0, 'name'
'milestone', 'creator', 'login'
```

A string-only syntax for paths (such as `a.b.c` or `a/b/c`) is not supported, since all conceivable syntaxes have drawbacks, and it is not up to `sanest` to make choices here.

Getting, setting, deleting. For getting, setting, and deleting items, paths can be used directly inside square brackets:

```
>>> d = sanest.dict(...)
>>> d['a', 'b', 'c'] = 123
>>> d['a', 'b', 'c']
123
>>> del d['a', 'b', 'c']
```

Alternatively, paths can be specified as a list or tuple instead of the inline syntax:

```
>>> path = ['a', 'b', 'c']
>>> d[path] = 123
>>> path = ('a', 'b', 'c')
>>> d[path]
123
```

Other operations. For the method based container operations taking a key or index, such as `sanest.dict.get()` or `sanest.dict.pop()`, paths must always be passed as a list or tuple:

```
>>> d.get(['a', 'b', 'c'], "default value")
```

Containment checks. The `in` operator that checks whether a key or index exists, also works with paths:

```
>>> ['milestone', 'creator', 'login'] in issue
True
>>> ['milestone', 'creator', 'xyz'] in issue
False
>>> ['labels', 0] in issue
True
>>> ['labels', 123] in issue
False
```

Automatic creation of nested structures. When setting a nested dictionary key that does not yet exist, the structure is automatically created by instantiating a fresh dictionary at each level of the path. This is sometimes known as *autovivification*:

```
>>> d = sanest.dict()
>>> d['a', 'b', 'c'] = 123
>>> d
sanest.dict({'a': {'b': {'c': 123}}})
>>> d.setdefault(['a', 'e', 'f'], 456)
456
>>> d
sanest.dict({'a': {'b': {'c': 123}, 'e': {'f': 456}}})
```

This only works for paths pointing to a dictionary key, not for lists (since padding with *None* values is seldom useful), but of course it will traverse existing lists just fine:

```
>>> d = sanest.dict({'items': [{'name': "a"}, {'name': "b"}]})
>>> d['items', 1, 'x', 'y', 'z'] = 123
>>> d['items', 1]
sanest.dict({'x': {'y': {'z': 123}}, 'name': 'b'})
```

CHAPTER 7

Type checking

In addition to the basic validation to ensure that all values adhere to the JSON data model, almost all `sanest.dict` and `sanest.list` operations support explicit *type checks*.

Getting, setting, deleting. For getting, setting, and deleting items, type checking uses slice syntax to indicate the expected data type:

```
>>> issue['id':int]
1
>>> issue['state':str]
'open'
```

Path lookups can be combined with type checking:

```
>>> issue['user', 'login':str]
'octocat'
>>> path = ['milestone', 'creator', 'id']
>>> issue[path:int]
1
```

Other operations. Other methods use a more conventional approach by accepting a *type* argument:

```
>>> issue.get('id', type=int)
1
>>> issue.get(['user', 'login'], type=str)
'octocat'
```

Containment checks. The `in` operator does not allow for slice syntax, so instead it uses a normal list with the type as the last item:

```
>>> ['id', int] in issue
True
>>> ['id', str] in issue
False
```

This also works with paths:

```
>>> ['user', 'login', str] in issue
True
>>> path = ['milestone', 'creator', 'id']
>>> [path, int] in issue
True
>>> [path, bool] in issue
False
```

Extended types. In its simplest form, the *type* argument is just the built-in type: `bool`, `float`, `int`, `str`, `dict`, `list`. This works well for simple types, but for containers, only specifying that the application 'expects a list' is often not good enough.

Typically lists are homogeneous, meaning that all values have the same type, and `sanest` can check this in one go. The syntax for checking the types of list values is a list containing a type, such as `[dict]` or `[str]`. For example, to ensure that a field contains a list of dictionaries:

```
>>> issue['labels':[dict]]
sanest.list([{"id": 208045946, "name": "bug"}])
```

To keep it sane, this approach cannot be used recursively, but then, nested lists are not that common anyway.

For dictionaries, `sanest` offers similar functionality. Its usefulness is limited, since it is not very common for dictionary values to all have the same type. (Note that dictionary keys are always strings.) The syntax is a literal dictionary with one key/value pair, in which the key is *always* the literal `str`, such as `{str: int}` or `{str: bool}`. For example, to ensure that all values in the dictionary pointed to by the path `'a', 'b', 'c'` are integers:

```
d['a', 'b', 'c':{str: int}]
```

Checking container values. To explicitly check that all values in a container have the same type, use `sanest.list.check_types()` or `sanest.dict.check_types()`, which take a *type* argument':

```
l = sanest.list()
l.append(1)
l.append(2)
l.append(3)
l.check_types(type=int)
```

This stand-alone check works for top-level containers, and can be used without combining it with another container operation. Sometimes, this may be more readable than a combined lookup and type check. For example:

```
>>> labels = issue['labels']
>>> labels.check_types(type=dict)
```

. . . may be more readable than:

```
>>> issue['labels':[dict]]
```

Type-safe iteration. Iterating over a list is a very common operation, and `sanest` makes it easy to do this in a type-safe way. One option is to check the types explicitly:

```
>>> labels = issue['labels']
>>> labels.check_types(type=dict)
>>> for label in labels:
...     pass
```

Another option is to use the `sanest.list.iter()` method:

```
>>> labels = issue['labels']
>>> for label in labels.iter(type=dict):
...     pass
```

A less readable but even shorter version would be:

```
>>> for label in issue['labels':[dict]]:
...     pass
```

For dictionaries with homogeneously typed values, similar features are available by using `sanest.dict.values()` or using `sanest.dict.items()`:

```
>>> d = sanest.dict(...)
>>> for value in d.values(type=int):
...     pass
>>> for key, value in d.items(type=int):
...     pass
```

CHAPTER 8

Wrapping

Both `sanest.dict` and `sanest.list` are thin wrappers around a regular `dict` or `list`, providing new features, but not changing the data structure in any way, which in practice means that the overhead of using `sanest` is relatively small. Internally, nested structures are just as they would be in regular Python. `sanest` adds a thin layer on top to provide a nice API to applications using it, and 'wraps' any container values it returns on the way out.

Wrapping existing containers. The `sanest.dict` and `sanest.list` constructors create a new container, and make a shallow copy when an existing `dict` or `list` is passed to it, analogous to the behaviour of the built-in `dict` and `list`.

`sanest` can also wrap an existing `dict` or `list` without making a copy, using the *classmethods* `sanest.dict.wrap()` and `sanest.list.wrap()`, that can be used as alternate constructors:

```
d = sanest.dict.wrap(existing_dict)
l = sanest.list.wrap(existing_list)
```

By default, this validates that the input matches the JSON data model. In some cases, for instance for just deserialised JSON data, these checks are not necessary, and can be skipped for performance reasons:

```
d = sanest.dict.wrap(existing_dict, check=False)
l = sanest.list.wrap(existing_list, check=False)
```

Unwrapping. The reverse process is *unwrapping*: to obtain a plain `dict` or `list`, use `sanest.dict.unwrap()` or `sanest.list.unwrap()`, which will return the original objects:

```
normal_dict = d.unwrap()
normal_list = l.unwrap()
```

Unwrapping is typically done at the end of a piece of code, when a regular `dict` or `list` is required, e.g. right before serialisation:

```
json.dumps(d.unwrap())
```

Unwrapping is a very cheap operation and does not make any copies.

Localised use. Wrapping an existing `dict` or `list` is also a very useful way to use `sanest` only in selected places in an application, e.g. in a function that modifies a regular `dict` that is passed to it, without any other part of the application being aware of `sanest` at all:

```
def set_fields(some_dict, num, flag):
    """
    Set a few fields in `some_dict`. This modifies `some_dict` in-place.
    """
    wrapped = sanest.dict.wrap(some_dict)
    wrapped["foo", "bar":int] = num * 2
    wrapped.setdefault(["x", "y", type=bool] = flag
```

CHAPTER 9

Error handling

Note: this section needs to be (re)written

When the data does not match what the code expects, `sanest` raises a sensible exception:

```
>>> print(d['users', 0, 'name':int])
Traceback (most recent call last):
...
InvalidValueError: expected int, got str at path ['users', 0, 'name']: 'alice'
```

Exceptions for missing data. to do

- `LookupError` (built-in)

  to do

  - `KeyError` (built-in)

    to do

  - `IndexError` (built-in)

    to do

Exceptions for problematic data. The following exceptions are raised for data that does not match what the code expects, and can be caught in the application. Instead of catching these exceptions, applications can also catch the built-in `ValueError`, in which case no `sanest` imports are needed.

- `ValueError` (built-in)

  - `sanest.DataError`

    Indicates problematic data. Never raised directly, but can be caught if the application does not care whether the source of the problem was an invalid structure or aen invalid value.

    * `sanest.InvalidStructureError`

      to do

    * `sanest.InvalidValueError`

      to do

Exceptions for problematic code. The following exceptions are typically the result of incorrect code, and hence should generally not be caught.

- `TypeError` (built-in)
  - `sanest.InvalidPathError`
  - `sanest.InvalidTypeError`

CHAPTER 10

API

`sanest.dict`

`sanest.list`

Exceptions

CHAPTER 11

Contributing

The source code and issue tracker for this package can be found on Github:

> https://github.com/wbolster/sanest

`sanest` has an extensive test suite that covers the complete code base. Please provide minimal examples to demonstrate potential problems.

CHAPTER 12

License