VFX Naming

Release 1.2.3-beta

Aug 19, 2023

Getting Started

CHAPTER 1

Installation

```
pip install vfxnaming
```

A complete suite of tools to manage naming conventions from one or more “Rule repositories”. Structure naming rules with your own custom tokens. Then use the library to solve names following those rules so your naming is consistent, and also to parse metadata from exisiting names (cus a name is basically a collection of metadata, right?)

CHAPTER 2

General Ideas

vfxnaming consists of three main entities: `Token`, `TokenNumber` and `Rule`. All of them are saved and read to and from a *repository* that contains their serialized versions.

A Rule is made of Tokens and hardcoded strings too. This is how a Rule pattern looks like. Values between curly braces '{ }' are Token placeholders.

Note: '{category}_{function}_{whatAffects}_{digits}_{type}'

CHAPTER 3

naming module

`vfxnaming.naming` is the main module of **vfxnaming**. Please refer to *Naming Module* for further details.

It consists of two key functions:

1. parse(path)
2. solve(args, kwargs)

And three working functions:

1. get_repo()
2. load_session()
3. save_session()

CHAPTER 4

logger module

`vfxnaming.logger` uses Python logging to track almost everything that's happening under the hood. This is really useful for debugging. Please refer to *Logger Module* for further details.

This is going to initialize the logger and output to stdout (console, terminal, etc)

```
from vfxnaming import logger
logger.init_logger()
```

This is going to initalize a log file where everything will be recorded.

```
from vfxnaming import logger
logger.init_file_logger()
```

CHAPTER 5

Credits

vfxnaming is completely based on Copyright (c) 2017 Cesar Saez work. I highly recommend his Website-Blog and the video tutorial series on his YouTube Channel

For more information and credits please check *Credits*

5.1 Naming Repository creation and loading

Using **vfxnaming** means basically creating a series of Tokens and Rules that then get saved to a repository (a simple folder with some files in it).

The saved files are the serialized objects in JSON format, which has the huge advantage of being interchangeable with practically any other programming language.

5.1.1 1.Repo Creation Session

```
import vfxnaming as n

n.add_token('whatAffects')
n.add_token_number('digits')
n.add_token(
    'category',
    natural='nat', practical='pra', dramatic='dra',
    volumetric='vol', default='nat'
)
n.add_token(
    'function',
    key='key', fill='fil', ambient='amb',
    bounce='bnc', rim='rim', kick='kik', custom='cst',
    default='cst'
)
n.add_token(
```

(continues on next page)

(continued from previous page)

```
    'type', lighting='LGT', animation='ANI', default='LGT'
)
n.add_rule(
    'lights',
    '{category}_{function}_{whatAffects}_{digits}_{type}'
)
my_repo = 'C:/path/to/my/repo'
n.save_session(my_repo)
```

This will result in the following files being created:

- lights.rule
- naming.conf
- category.token
- function.token
- digits.token
- type.token
- whatAffects.token

If there is only one rule, it'll be set as active by default. If there is more than one, you need to activate that template before using parsing or solving.

When saving the session, all Tokens and Rules in memory will be saved to the repository along with a naming.conf file that stores the last active Rule (It'll be set as active again when loading the session from the repo next time.)

5.1.2 1.1 Adding Tokens

```
1 n.add_token('whatAffects')
2 n.add_token_number('digits')
3 n.add_token(
4     'category',
5     natural='nat', practical='pra', dramatic='dra',
6     volumetric='vol', default='nat'
7 )
```

In line 1 we're creating a **Required Token**. This means that for solving the user has to provide a value. This is a explicit solve.

In line 2 we're creating a **Number Token**. This is a special Token really useful for working with version like or counting parts of a name. It's always required.

In line 3 we're creating an **Optional Token**, which means that for solving the user can pass one of the options in the Token or simply ignore passing a value and the Token will solve to it's default option. This is an implicit solve, which helps to greatly reduce the amount of info that needs to be passed to solve for certain cases.

For more information on implicit and explicit solving please check *Solving*

5.1.3 1.2 Adding Rules

```
n.add_rule(
    'lights',
    '{category}_{function}_{whatAffects}_{digits}_{type}'
)

n.add_rule(
    'filename',
    'crazy_hardcoded_value_{awesometoken}',
    n.Rule.ANCHOR_END
)
```

Here we're creating naming rules, giving them a name, a pattern and an anchor optionally. *Name must be unique* for each rule in the repo.

Patterns must be structured so that each Token is identified by it's name and enclosed between curly brackets '{ }'.

Anchoring means you can force the evaluation of your Rule to be from left to right (default) or right to left or both. Really useful when you have hardcorded values in your naming Rule. Options for anchoring: Rule.ANCHOR_START (default), Rule.ANCHOR_END, Rule.ANCHOR_BOTH

5.1.4 2. Repo Loading Session

These files can then be read from next time you need to use your new naming rules by passing the repo location to the load_session function. Python object instances will be loaded into memory and you'll be able to interact with them (solving, parsing, adding new rules, new tokens, etc):

```
import vfxnaming as n

my_repo = 'C:/path/to/my/repo'
n.load_session(my_repo)

all_rules = n.get_rules()
all_tokens = n.get_tokens()
```

Warning: It's important to manipulate both Tokens and Rules through their module functions, not the object methods. This is so the system can keep track of what's created, removed, updated, etc, during the repo creation session.

5.2 Solving

Solving from a Rule means passing it some parameters and getting back a *name* which follows the Rule's pattern.

Note: The solving function is vfxnaming.solve(args, kwargs)

Let's set these Tokens and Rules.

```
import vfxnaming as n

# CREATE TOKENS
n.add_token('whatAffects')
```

(continues on next page)

(continued from previous page)

```
n.add_token_number('digits')
n.add_token(
    'category',
    natural='nat', practical='pra', dramatic='dra',
    volumetric='vol', default='nat'
)
n.add_token(
    'function',
    key='key', fill='fil', ambient='amb',
    bounce='bnc', rim='rim', kick='kik', custom='cst',
    default='custom'
)
n.add_token(
    'type', lighting='LGT', animation='ANI', default='LGT'
)

# CREATE RULES
n.add_rule(
    'lights',
    '{category}_{function}_{whatAffects}_{digits}_{type}'
)

n.set_active_rule("lights")
```

5.2.1 Explicit Vs. Implicit

It would not make any sense to make the user pass each and every Token all the time to be able to solve for a name. That'd be the equivalent, almost, to typing the name by hand. Also, it'd be good if the user doesn't have to know all token names by heart (though Rule.fields can help you with that).

That's why vfxnaming.solve() accepts both args and kwargs. Not only that, but if given Token is optional and you want to use it's default value, you don't need to pass it at all.

```
n.solve(
    category='natural', function='custom',
    whatAffects='chars', digits=1, type='lighting'
)
n.solve(whatAffects='chars', digits=1)
n.solve('chars', 1)
```

Each of these calls to vfxnaming.solve() will produce the exact same result:

Note: natural_custom_chars_001_LGT

If you don't pass a required Token (either as an argument or keyword argument), such as 'whatAffects' in this example, you'll get a **TokenError**. You'll also get a **TokenError** if you try to parse a value that doesn't match any of the options in the Token.

5.2.2 Solving rules with repeated tokens

If your rule uses the same token more than once, then the library will handle it by adding an incremental digit to the token name when parsing and solving.

Here is an example of such a rule being created.

```
import vfxnaming as n

n.add_token(
    'side', center='C',
    left='L', right='R',
    default='C'
)
n.add_token(
    'region', orbital="ORBI",
    parotidmasseter="PAROT", mental="MENT",
    frontal="FRONT", zygomatic="ZYGO",
    retromandibularfossa="RETMAND"
)
n.add_rule(
    "filename",
    '{side}-{region}_{side}-{region}_{side}-{region}'
)

n.save_session()
```

When **Solving** a name for a rule with repeated tokens you have three options:

1. Explicitly pass each repetition with an added digit for each repetition

```
n.solve(
    side1="center", side2="left", side3="right",
    region1="mental", region2="parotidmasseter",
    region3="retromandibularfossa"
)
```

2. Explicitly pass some of the repetitions with an added digit for each one. The ones you didn't pass are going to use the Token's default.

```
n.solve(
    side1="center", side3="right",
    region2="parotidmasseter",
    region3="retromandibularfossa"
)
```

3. Explicitly pass just one argument, with no digit added. Your argument will be used for all token repetitions.

```
n.solve(
    side="left",
    region1="mental", region2="parotidmasseter",
    region3="retromandibularfossa"
)
```

5.3 Parsing

Names contain a bunch of metadata about what they are. All of this metadata can be read and used to our advantage. Each Token is basically a piece of metadata. Each Rule helps us extract that metadata from names.

Note: The parsing function is vfxnaming.parse(name)

Warning: The appropiate Rule must be set as active before calling the parse() function. Use vfxnaming.set_active_rule("rule_name")

Let's set these Tokens and Rules.

```
import vfxnaming as n

# CREATE TOKENS
n.add_token('whatAffects')
n.add_token_number('digits')
n.add_token(
    'category',
    natural='nat', practical='pra', dramatic='dra',
    volumetric='vol', default='nat'
)
n.add_token(
    'function',
    key='key', fill='fil', ambient='amb',
    bounce='bnc', rim='rim', kick='kik', custom='cst',
    default='cst'
)
n.add_token(
    'type', lighting='LGT', animation='ANI', default='LGT'
)

# CREATE RULES
n.add_rule(
    'lights',
    '{category}_{function}_{whatAffects}_{digits}_{type}'
)

n.set_active_rule("lights")
```

And then let's parse this name:

```
n.parse("dramatic_bounce_chars_001_LGT")
```

The result will be the following dictionary with all the metadata extracted to key, value pairs:

```
result = {
    "category": "dramatic",
    "function": "bounce",
    "whatAffects": "chars",
    "digits": 1,
    "type":"lighting"
}
```

5.3.1 Parsing rules with repeated tokens

If your rule uses the same token more than once, then the library will handle it by adding an incremental digit to the token name when parsing and solving.

Here is an example of such a rule being created.

```
import vfxnaming as n

n.add_token(
    'side',
    center='C', left='L', right='R',
    default='C'
)
n.add_token(
    'region',
    orbital="ORBI", parotidmasseter="PAROT", mental="MENT",
    frontal="FRONT", zygomatic="ZYGO", retromandibularfossa="RETMAND"
)
n.add_rule(
    "filename",
    '{side}-{region}_{side}-{region}_{side}-{region}'
)

n.save_session()
```

When **Parsing** metadata using a rule with repeated tokens, the dictionary you get back will have the keys for the repeated Token altered by an incremental digit at the end of the token name.

```
result = {
    "side1": "center", "region1": "frontal",
    "side2": "left", "region2": "orbital",
    "side3": "right", "region3": "zygomatic"
}
```

There are many ways to substract that digit from the keys, but maybe the most reliable could be to use regular expressions. You can also use the `rule.fields` attribute and compare your keys to the pure Token name.

```
import re

pattern = re.compile(r'[a-zA-Z]+')
for key in result.keys():
    print(pattern.search(key))
```

5.4 Naming Module

vfxnaming.naming.**get_repo**()
: Get repository location from either global environment variable or local user, giving priority to environment variable.

  Environment varialble name: NAMING_REPO

  Returns: str: Naming repository location

vfxnaming.naming.**load_session**(*repo=None*)
: Load rules, tokens and config from a repository, and create Python objects in memory to work with them.

  Args: repo (str, optional): Absolute path to a repository. Defaults to None.

  Returns: bool: True if loading session operation was successful.

vfxnaming.naming.**parse**(*name*)
: Get metadata from a name string recognized by the currently active rule.

  -For rules with repeated tokens:

If your rule uses the same token more than once, the returned dictionary keys will have the token name and an incremental digit next to them so they can be differentiated.

Args: name (str): Name string e.g.: C_helmet_001_MSH

Returns: dict: A dictionary with keys as tokens and values as given name parts. e.g.: {'side':'C', 'part':'helmet', 'number': 1, 'type':'MSH'}

vfxnaming.naming.**save_session**(*repo=None*)

Save rules, tokens and config files to the repository.

Raises: IOError, OSError: Repository directory could not be created.

Args: repo (str, optional): Absolue path to a repository. Defaults to None.

Returns: bool: True if saving session operation was successful.

vfxnaming.naming.**solve**(**args*, ***kwargs*)

Given arguments are used to build a name following currently active rule.

-For rules with repeated tokens:

If your rule uses the same token more than once, pass arguments with the token name and add an incremental digit

i.e.: side1='C', side2='R'

If your rule uses the same token more than once, you can also pass a single instance of the argument and it'll be applied to all repetitions.

i.e.: side='C'

If your rule uses the same token more than once, you can ignore one of the repetitions, and the solver will use the default value for that token.

i.e.: side1='C', side4='L'

Raises: SolvingError: A required token was passed as None to keyword arguments. SolvingError: Missing argument for one field in currently active rule.

Returns: str: A string with the resulting name.

5.5 Rules module

Adding a Rule is as simple as using the `add_rule()` function and passing a name and a pattern which uses the Tokens you have created.

The **name** should be unique in the repository.

The **pattern** is a simple string where you place the names of the Tokens you'd like to use inbetween curly brackets and separate them using any of the most commonly separators (hyphen, underscore, dot, etc)

Then there is also the option to use **Anchoring**. This means you can force the evaluation of your Rule to be from left to right (default) or right to left or both. Really useful when you have hardcorded values in your naming Rule. Options for anchoring: Rule.ANCHOR_START (default), Rule.ANCHOR_END, Rule.ANCHOR_BOTH

```
import vfxnaming as n

n.add_rule(
    'lights',
    '{category}_{function}_{whatAffects}_{digits}_{type}'
)
```

(continues on next page)

(continued from previous page)

```
n.add_rule(
    'filename',
    '{side}-{region}_{side}-{region}_{side}-{region}'
)

n.add_rule(
    'filename',
    'crazy_hardcoded_value_{awesometoken}',
    n.Rule.ANCHOR_END
)
```

class vfxnaming.rules.**Rule**(*name*, *pattern*, *anchor=1*)

Each rule is managed by an instance of this class. Fields exist for each Token and Separator used in the rule definition.

Args: name (str): Name that best describes the rule, this will be used as a way to query the Rule object.

pattern (str): The template pattern to use, which uses existing Tokens. e.g.: '{side}_{region}_{side}_{region}.png'

anchor: ([Rule.ANCHOR_START, Rule.ANCHOR_END, Rule.ANCHOR_BOTH], optional): For parsing, regex matching will look for a match from this Anchor. If a pattern is anchored to the start, it requires the start of a passed path to match the pattern. Defaults to ANCHOR_START.

data()

Collect all data for this object instance.

Returns: dict: {attribute:value}

fields

Returns: [tuple]: Tuple of all Tokens found in this Rule's pattern

classmethod from_data(*data*)

Create object instance from give data. Used by Rule, Token, Separator to create object instances from disk saved data.

Args: data (dict): {attribute:value}

Returns: Serializable: Object instance for Rule, Token or Separator.

parse(*name*)

Build and return dictionary with keys as tokens and values as given names.

If your rule uses the same token more than once, the returned dictionary keys will have the token name and an incremental digit next to them so they can be differentiated.

Args: name (str): Name string e.g.: C_helmet_001_MSH

Returns: dict: A dictionary with keys as tokens and values as given name parts. e.g.: {'side':'C', 'part':'helmet', 'number': 1, 'type':'MSH'}

regex

Returns: [str]: Regular expression used to parse from this Rule

solve(****values**)

Given arguments are used to build a name.

Raises: SolvingError: Arguments passed do not match with rule fields.

Returns: str: A string with the resulting name.

vfxnaming.rules.**add_rule**(*name*, *pattern*, *anchor=1*)
Add rule to current naming session. If no active rule is found, it adds the created one as active by default.

Args: name (str): Name that best describes the rule, this will be used as a way to invoke the Rule object.

pattern (str): The template pattern to use, which uses existing Tokens. e.g.: '{side}_{region}_{side}_{region}.png'

anchor: ([Rule.ANCHOR_START, Rule.ANCHOR_END, Rule.ANCHOR_BOTH], optional): For parsing, regex matching will look for a match from this Anchor. If a pattern is anchored to the start, it requires the start of a passed path to match the pattern. Defaults to ANCHOR_START.

Returns: Rule: The Rule object instance created for given name and fields.

vfxnaming.rules.**get_active_rule**()
Get currently active rule for the session. This is the rule that will be used to parse and solve from.

Returns: Rule: Rule object instance for currently active Rule.

vfxnaming.rules.**get_rule**(*name*)
Gets Rule object with given name.

Args: name (str): The name of the rule to query.

Returns: Rule: Rule object instance for given name.

vfxnaming.rules.**get_rules**()
Get all Rule objects for current session.

Returns: dict: {rule_name:Rule}

vfxnaming.rules.**has_rule**(*name*)
Test if current session has a rule with given name.

Args: name (str): The name of the rule to be looked for.

Returns: bool: True if rule with given name exists in current session, False otherwise.

vfxnaming.rules.**load_rule**(*filepath*)
Load rule from given location and create Rule object in memory to work with it.

Args: filepath (str): Path to existing .rule file location

Returns: bool: True if successful, False if .rule wasn't found.

vfxnaming.rules.**remove_rule**(*name*)
Remove Rule from current session.

Args: name (str): The name of the rule to be removed.

Returns: bool: True if successful, False if a rule name was not found.

vfxnaming.rules.**reset_rules**()
Clears all rules created for current session.

Returns: bool: True if clearing was successful.

vfxnaming.rules.**save_rule**(*name*, *directory*)
Saves given rule serialized to specified location.

Args: name (str): The name of the rule to be saved.

directory (str): Path location to save the rule.

Returns: bool: True if successful, False if rule wasn't found in current session.

vfxnaming.rules.**set_active_rule**(*name*)

Sets given rule as active for the session. This it the rule that's being used to parse and solve from.

Args: name (str): The name of the rule to be activated.

Returns: bool: True if successful, False otherwise.

5.6 Tokens module

Tokens are the meaningful parts of a template. A token can be required, meaning fully typed by the user, or can have a set of options preconfigured.

If options are present, then one of them is the default one. Each option follows a {full_name:abbreviation} schema, so that names can be short but meaning can be recovered easily. The default option might be passed explicitly by the user by passing a *default* argument (it must match one of the options in the Token). If no default options is explicitly passed, the Token will sort options alphabetically and pick the first one. Please notice if you pass the *default* option explicitly, you can use the abbreviation or the full option name.

```
1 n.add_token('whatAffects')
2 n.add_token_number('digits')
3 n.add_token('category', natural='nat',
4             practical='pra', dramatic='dra',
5             volumetric='vol', default='nat')
```

In line 1 we're creating a **Required Token**. This means that for solving the user must provide a value. This is a explicit solve.

In line 2 we're creating a **Number Token**. This is a special Token really useful for working with version like or counting parts of a name. It's always required.

In line 3 we're creating an **Optional Token**, which means that for solving the user can pass one of the options in the Token or simply ignore passing a value and the Token will solve to it's default option. This is an implicit solve, which helps to greatly reduce the amount of info that needs to be passed to solve for certain cases.

For more information on implicit and explicit solving please check *Solving*

vfxnaming.tokens.**add_token**(*name*, ***kwargs*)

Add token to current naming session. If 'default' keyword argument is found, set it as default for the token instance.

Args: name (str): Name that best describes the token, this will be used as a way to invoke the Token object.

kwargs: Each argument following the name is treated as an option for the new Token.

Raises: TokenError: If explicitly passed default does not match with any of the options.

Returns: Token: The Token object instance created for given name and fields.

vfxnaming.tokens.**add_token_number**(*name*, *prefix=''*, *suffix=''*, *padding=3*)

Add token number to current naming session.

Args: name (str): Name that best describes the token, this will be used as a way to invoke the TokenNumber object.

prefix (str, optional): Prefix for token number. Useful if you have to use 'v' as prefix for versioning for example.

suffix (str, optional): Suffix for token number.

padding (int, optional): Padding a numeric value with this leading number of zeroes. e.g.: 25 with padding 4 would be 0025

Returns: TokenNumber: The TokenNumber object instance created for given name and fields.

vfxnaming.tokens.**get_token**(*name*)
Gets Token or TokenNumber object with given name.

Args: name (str): The name of the token to query.

Returns: Rule: Token object instance for given name.

vfxnaming.tokens.**get_tokens**()
Get all Token and TokenNumber objects for current session.

Returns: dict: {token_name:token_object}

vfxnaming.tokens.**has_token**(*name*)
Test if current session has a token with given name.

Args: name (str): The name of the token to be looked for.

Returns: bool: True if rule with given name exists in current session, False otherwise.

vfxnaming.tokens.**load_token**(*filepath*)
Load token from given location and create Token or TokenNumber object in memory to work with it.

Args: filepath (str): Path to existing .token file location

Returns: bool: True if successful, False if .token wasn't found.

vfxnaming.tokens.**remove_token**(*name*)
Remove Token or TokenNumber from current session.

Args: name (str): The name of the token to be removed.

Returns: bool: True if successful, False if a rule name was not found.

vfxnaming.tokens.**reset_tokens**()
Clears all rules created for current session.

Returns: bool: True if clearing was successful.

vfxnaming.tokens.**save_token**(*name*, *directory*)
Saves given token serialized to specified location.

Args: name (str): The name of the token to be saved. filepath (str): Path location to save the token.

Returns: bool: True if successful, False if rule wasn't found in current session.

5.7 Logger Module

5.8 Changelog

5.8.1 1.2.4-beta

Bug fixes: -Fix creation of logging handlers

5.8.2 1.2.3-beta

Bug fixes: -Checks for naming.conf existance when loading a repo and resets both rules and tokens

5.8.3 1.2.2-beta

Improvements: -Added remove_option(), update_option(), has_option_fullname(), has_option_abbreviation() methods to Token

5.8.4 1.2.1-beta

Changes:

- Default option when adding Tokens must now be one the options passed for the Token.
- Default option can now be explicitly passed as the full name option or its abbreviation.
- When Parsing, if no value matches with the Token options and Token is Optional, raise TokenError.
- IMPORTANT NOTE: This version might require updates on old Tokens that accepted any default value before. Quick and simple fix is to add that value as an option for the Token.

5.8.5 1.2.0-beta

Features:

- Adds anchoring, so each Rule may be able to parse and solve from right to left, or left to right. Useful for Rules with hardcoded values (not only Tokens)
- Solving and parsing now use Regular Expressions under the hood.
- A lot of updates to docs.

Changes:

- Removes Separator entity
- IMPORTANT NOTE: This version is not backwards compatible

5.8.6 1.1.6-beta

Features:

- Adds support for parsing and solving repeated tokens within the same rule.

Improvements:

- Added documentation for the entire package
- Added custom errors

5.8.7 1.0.0-alpha

Features:

- Parsing of metadata from names (Including version like strings with digits)
- Solving names using pre-established Rules and Tokens
- Repository saving and loading of serialized Rules and Tokens in json format

5.9 Roadmap

-Implement nested Tokens

5.10 Credits

5.10.1 Chris Granados - Xian

I'm the author and mantainer of **vfxnaming**.

5.10.2 Cesar Saez

vfxnaming is completely based on Copyright (c) 2017 Cesar Saez work. I highly recommend his Website-Blog and the video tutorial series on his YouTube Channel

-Why no fork from Cesar's Repo?

> I found myself using and modifying his code to fit my needs. Not only that, but he has an AMAZING video tutorial series on this topic and my code deviated a bit too much from what he shows in the videos.

-What are the main differences from Cesar's?

1. Implemented a special Token for numbers with the ability to handle pure digits and version like strings (e.g.: v0025) with padding settings.
2. Implemented Separators. Used to be a completely separate object in older versions, but now separators are simply inferred from the regular expressions used.
3. Switched the entire test suite to pytest which is what I use for standalone stuff.
4. Refactored the code to make it a bit more modular. Which in turn makes it less portable, but it was getting too long otherwise with my modifications. To compensate for portability, it's pip installable now.

5.10.3 Martin Pengelly-Phillips

The regular expressions logic was taken in part from the work Martin Pengelly-Phillips did on Lucidity. Also tweaked and modified a lot to fit my needs, so don't expect the same Lucidity behaviour.

Python Module Index

v

Index