Improv Documentation

Release 0.6.0

Bruno Dias

Jun 19, 2017

Contents

Contents:

CHAPTER 1

Introduction

Improv is a javascript library for procedurally generating text, designed primarily to be used in games and interactive fiction. It's similar in function to Kate Compton's Tracery; it generates text at random from a tree-like structure of "snippets" that can be composed together to form a template.

Improv, however, supports the use of a *world model* and *filters* that can dynamically change which phrases are considered for inclusion in the text, as it is being generated. This means Improv specs can be written to construct coherent text in relatively free-form ways, ensuring agreement with itself or with an external world model.

> The HMS Reliable is a clipper commissioned 6 years ago.
>
> Using a whale oil engine, she can reach speeds upwards of 32 knots. The Reliable is one of the new generation of vessels built to fight against the Arkodian fleet in the Short War. Her crew is known to be one of the more disciplined in the Navy. She is currently serving as a colonial troop transport.

The preceding text was procedurally generated by Improv's `hms.js` demo. While the statements about the fictional ship are arranged in a largely random fashion, Improv ensures that they agree with a number of rules:

- Only Navy ships have the prefix "HMS"
- "Reliable" is a name reserved for clippers and other freighters
- "Whale oil engines" only occur in newer ships
- Only newer ships with engines are so fast
- Only freighters are assigned as troop transports
- Don't repeat phrases
- Try to mention different aspects of the ship with each phrase

All of those rules are expressed only through the dataset Improv uses to generate text; no actual code is involved in expressing them. This means Improv can deal with relatively complex realities without the need of elaborate logic.

CHAPTER 2

Usage

Installation

Improv is an npm module, meant to be used both on Node.js and on browser environments through the use of a module bundler such as Browserify or Webpack. Adding Improv to your project is therefore done in the usual way with *npm install –save improv*.

Quick Example

A brief usage example:

```
const Improv = require('improv'); // import Improv from 'improv';

const spec = {
  animal: {
    groups: [
      {
        tags: [['class', 'mammal']],
        phrases: ['dog', 'cat']
      },
      {
        tags: [['class', 'bird']],
        phrases: ['parrot']
      }
    ]
  },
  root: {
    groups: [
      {
        tags: [],
        phrases: [
          "[name]: I have a [:animal] who is [#2-7] years old."
        ]
      }
    ]
  }
};

const improv = new Improv(spec, {
  filters: [Improv.filters.mismatchFilter()]
});

const bob = { name: 'Bob' };
const alice = { name: 'Alice', tags: [['class', 'mammal']] };
const carol = { name: 'Carol', tags: [['class', 'bird']] };

const lines = [
  improv.gen('root', bob),
  improv.gen('root', alice),
  improv.gen('root', carol)
];

console.log(lines.join('\n'));
```

This script, when run, should produce something like:

> Bob: I have a dog who is 6 years old.
> Alice: I have a cat who is 4 years old.
> Carol: I have a parrot who is 7 years old.

The output isn't completely random; it obeys certain rules. Alice always has a dog or a cat; Carol always has a parrot. Bob might have either animal. This is simplistic, but this type of world model can be composed to express fairly complex rules.

This example be found in the improv source distribution, under `demo/pets.js`. Another, more elaborate example is also included, `demo/hms.js`. You can build both demos by doing `gulp demo` from the root folder of the source distribution; this should transpile them and make it possible to run them by doing `node demo_build/pets.js`.

Read the API documentation for information on how to create Improv generators and use them.

CHAPTER 3

Improv API Reference

Constructor

class Improv (*spec*, *options*)

Arguments

- **spec** (*object*) – The spec object containing a set of snippets.
- **options** (*object*) – An options object.

Returns An Improv generator.

Creates a generator according to the given spec. The options object is used to set its behavior.

Spec

The spec is a plain javascript object that defines all of the snippets that the generator has access to. It should follow this structure:

```
{
  snippetName: {
    groups: [
      {
        tags: [['individual', 'tag']],
        phrases: ['individual phrase']
      }
    ]
  }
}
```

That is, a plain object with keys corresponding to the names of snippets and values corresponding to snippet objects. Each snippet object has one property, `groups`, which itself is an array of group objects. Group objects have two properties, `tags`` (an array of arrays, where each array is an individual tag) and ``phrases` (an array of strings, with each string being a potential phrase that the snippet can produce).

Binding

Optionally, a snippet object can have it's `bind` property set to `true`. When the generator is run, that snippet will be chosen randomly only the first time it is called; afterwards, the output of that snippet (including any template directives) will be resolved and "frozen." This is useful, for instance, if you want the name of something to be consistent in the text but you're not sure when or where that name will show up first and have to be generated.

Note that binding is done *per model*; bindings are added to the model as properties of a `bindings` object.

Options

The options object defines the behavior of the generator. The default options are:

```
{
  filters: [],
  reincorporate: false,
  persistence: true,
  salienceFormula: function (a) { return a; },
  audit: false,
  rng: undefined,
  builtins: {},
  submodeler: function () { return {}; }
}
```

Filters

The `filters` option (default: empty array) should be an Array of functions that are applied as filters to select and rank phrases; see the filter API documentation.

Reincorporation

The `reincorporate` option (default: false) defines whether or not the generator should reincorporate used tags. If set to true, the generator will mutate given model objects, merging any tags of used phrases with the object's own tags. This is not done by simply concatenating the model's tags with the phrase's tags. Rather, any corresponding tags in the model are updated to reflect the tags in the phrase (where "corresponding" means the the first element in the tag is the same), while new tags are added to the end of the model's tag list.

Reincorporation means that Improv can make decisions about the model it's working with and that those decisions will be referenced by future phrase generation.

Persistence

The `persistence` option (default: true) defines whether the generator will save history and tag history.

Salience formula

The `salienceFormula` option (default: identity function) is a callback that is used to calculate the salience threshold. After Improv applies filters and removes inappropriate phrases from the list of potential phrases, it then calculates the maximum salience score among possible phrases. This value is passed to `salienceFormula`, and the return value from that function is used as the salience threshold. Improv will only use phrases with a salience score equal to or greater than the salience threshold.

What this means is that, by default, Improv will use only the phrases that score highest on salience, as given by the filters that it applied. Using a different formula allows for fuzzier ranking of phrases, or simply slightly more randomness in Improv's phrase choice.

Auditing

The `audit` option (default: false) sets whether or not to turn on audit logging for this generator. Audit logging is done without any regard for memory or processing efficiency; it's meant as a tool to help you find "lumps" in your generator corpora (ie, things that come up too often or too rarely, biases and so on) and so it slows things down significantly.

Currently, the following audit data is available:

`Improv.prototype.`**`phraseAudit`**

A property of generator objects with audit turned on. The phrase audit is a Map object where the keys are strings (snippet names) and the values are themselves Maps. Each inner map has a key for each phrase that is a valid result for the snippet, and a value of an integer that corresponds to how often this phrase was used. This map is supposed to be comprehensive, meaning that phrases that don't show up at all will be there, with a value of zero.

This tally is run regardless of history saving. The intention is that you can run your generator thousands of times then dump this map data to whatever data format you prefer and look at the aggregate results to see if there are phrases that are never being used (because their salience is always too low), phrases that come up disproportionately often, and so on.

Using a custom RNG

The `rng` option (default: undefined) allows for supplying a custom random number generator, for instance if you want to use a seeded generator or if, for some deranged reason, you want your random text generated with cryptographically secure pseudorandom numbers.

The rng should be a function that supplies the same interface as Math.random(), that is, it should return a floating-point number between 0 (inclusive) and 1 (exclusive). When the generator object is created, the function is bound to it, so inside the rng function, `this` refers to the generator itself.

Builtins

See the section on *Templating*.

Submodeler

See the section on *Submodels*.

Methods

`Improv#gen`(*snippet*[, *model = {}*])

Arguments

- **`snippet`** (*`string`*) – The name of the snippet to be generated.
- **`model`** (*`object`*) – A model object.

Generates text according to a given snippet. Returns the generated text. Note that this is **not** a pure function; it can mutate the model object, attaching bindings to it and tags (if reincorporation is turned on).

`Improv#clearHistory``()`

Clears the generator's phrase history.

`Improv#clearTagHistory``()`

Clears the generator's tag history.

CHAPTER 4

Filtering API

The filtering API is the backbone of Improv's world model; it allows for groups of phrases to be selected according to certain rules.

A filter is just a function that supplies a specific API:

`filter`(*group*, *model*)

Arguments

- **`group`** (*`object`*) – The group of phrases being considered.
- **`model`** (*`object`*) – The model object being used.

The model object in this instance is the same one passed to *`Improv#gen()`*, and the group object is one of the groups included in the group list for the given snippet, in the spec used to create the generator.

When *`Improv#gen()`* is called, the generator's filters are called in order on each group of phrases to be considered, to calculate that group's salience score. Note that, if a filter earlier in the generator's filter list has excluded a group, then that group will never be passed to subsequent filters.

A filter is supposed to return one of three kinds of values:

- A number, which is treated as a score offset; it's added to the score for the group being tested.
- `null`, in which case the group is excluded from the list and won't be passed to other filters or used.
- An Array. In this case, element 0 of the array is treated as a score offset, while element 1 is treated as a modified group that is to be used instead of the original group. Note that this should be a new object, to avoid mutating the original group!

Standard filters

The object Improv.filters, imported as part of Improv, includes a number of standard filters. Note that the attributes of Improv.filters are *factories* that return a filter callback, not filters themselves. So to define a generator with all four standard filters:

```
const generator = new Improv(spec, {
  filters: [
    mismatchFilter(),
    partialBonus(),
    fullBonus(),
    dryness()
  ]
});
```

Note that `mismatchFilter()` itself returns a callback that is used as the actual filter in this instance. For the sake of consistency, every property of Improv.filters is one such factory, even if they don't take options.

`Improv.filters.`**`mismatchFilter`**`()`

The mismatch filter excludes from consideration (returns `null`) any group that has a tag which is mismatched against the model. What this means is that the tags are a matched pair (they have the same first element), but they different subsequent elements. `['animal', 'dog']` and `['animal', 'cat']` are mismatched.

This filter can be used to prevent contradictory statements; in the `hms.js` demo, for instance, ships can either be sailing vessels or have engines, and some ships with engines are electric while others use a combustion engine; the mismatch filter ensures that statements that apply to an electric vessel won't be used for a sailing vessel, while some statements can apply to both kinds of vessels with engines.

`Improv.filters.`**`partialBonus`**`(`[*bonus = 1*, *cumulative = false*]`)`

Arguments

- **`bonus`** (*number*) – The bonus to be given for matches.

Paran boolean cumulative Whether to apply that bonus for every match found.

A partial match is a set of two tags where one tag is shorter, but all elements of the shorter tag match their corresponding elements in the longer tag; this filter gives a salience score bonus to groups that have at least one partial match with the model's tags. If `cumulative` is true, this bonus is multiplied by the number of matches.

`Improv.filters.`**`fullBonus`**`(`[*bonus = 1*, *cumulative = false*]`)`

Identical to *`Improv.filters.partialBonus()`*, except it gives a bonus to full matches. A full match is a set of two tags where both tags are identical.

`Improv.filters.`**`dryness`**`()`

Removes phrases that are already present in the generator's history. DRY in this instance stands for "don't repeat yourself." Keep in mind that when using the dryness filter, it's important to carefully design the corpus and program so that Improv doesn't run out of valid phrases to say.

CHAPTER 5

Templating

Improv includes some basic templating functionality. For the most part, this is used so that snippets can be nested, allowing one phrase to include the output from a different snippet.

Warning: Improv's templating engine is very simple, and it doesn't do any sort of validation or checking. As a result, do not pass "dirty" user-submitted data into Improv; there is no guarantee that this is safe.

When Improv generates a phrase, that phrase gets run by the templating engine, from the start to the end of the phrase. It leaves everything alone other than *directives*, substrings wrapped in [brackets]. Directives are substituted in place by their results.

Directives are resolved *before* the rest of the phrase is parsed. This means that if you write a phrase thusly:

```
'[:statement] [:statement] [:statement] [:statement]'
```

Each subsequent `[:statement]` directive will be run in order. So if reincorporation is on for this generator, subsequent statements will "see" the changes to the model produced by earlier ones. This makes it relatively easy to build paragraphs out of a single snippet that calls several other snippets.

Literal Directive

Directives wrapped in `'single quotes'` will be treated as a string literal. That is, they will produce whatever text is inside the quotes. Ordinarily, this is useless, but it can be used to pass a literal text argument into a chained directive function.

Property Directive

Directives that don't start with a special character (# or :) will produce a stringified form of the model's property corresponding to the directive's text. So for example, `[name]` will look for a property in the model object called "name", and substitute in the content of that.

You can look up nested properties using the normal dot notation: `[name.full]` for example. Note that if one of the intervening properties is undefined, this will throw an error, and it can have unexpected results if the property does not convert gracefully into a string.

Number Directive

Directives starting with a `#`, such as `[#1-20]`, will produce a random (Integer) number. The content of the directive should be two integers separated by a hyphen (`-`); the number will be between the first and the second one, inclusive.

Recursion Directive

Directives starting with `:` will treat the directive text as a snippet to be called from the same generator. This allows composing snippets together, so that a single snippet can be used to generate an entire paragraph or description of something.

Note that by the time this recursion happens and the "inner snippet" is called, the phrase for the "outer snippet" has already been selected, and so have phrases for any preceding snippet. When using reincorporation and filters, this means that decisions about the model are made roughly in the same order as the text.

Chained Directives

Directives that have spaces in them (other than trailing or leading spaces) will be treated as a chain of methods to be called in order from the model object. Which is to say, this: `[funcA funcB propA]` will produce output equivalent to this: `model.funcA(model.funcB(model.propA))`. This allows endowing a model with functions to transform text and applying them from within templates.

Note that each individual word inside the directive is, itself, treated as an individual directive and expanded, so it is possible to use `:`, for instance.

The templating module includes a handful of built-in functions that supercede the model's own methods and can be used as linguistic conveniences.

- `cap` capitalises the first character in text.
- `a` will prefix the text with either "a" or "an", roughly in accordance with normal English grammar.
- `an` is an alias for `a`.
- `An` and `A` are equivalent to `cap a`.

You can therefore write phrases such as `In this county, it is forbidden to sell [an :animal] for less than twelve shillings.` and get appropriate results: "a dog", but "an aardvark."

Note that the algorithm currently used to produce a/an is fairly naive, and doesn't take into consideration most corner cases; it assumes that words starting in a, e, i, or o all take "an", while words starting in other letters take "a."

You can add your own builtins to a generator by specifying the `builtins` property in the generator options, as an object. For example:

```
const generator = new Improv(snippetData, {
  builtins: {
  upcap (str) { return str.toUpperCase(); }
  }
});
```

Will allow you to call `upcap` as a built-in function from within templates.

CHAPTER 6

Submodels

Submodels are a powerful feature that allows Improv models to be nested. For example, a model representing a story could contain several underlying submodels describing individual character roles within that story. Each submodel maintains its own independent set of tags and bindings.

To use a submodel, use the `[>submodel:snippet]` templating directive. The directive starts with a `>`, followed by the name of the submodel, followed by a `:`,followed by the name of the snippet to generate using that submodel.

Programmatically, submodels are just own properties of models that happen to be objects. You can populate a model with submodels ahead of time, if you want; otherwise, Improv will create them as necessary.

Submodelers

The submodeler is the function used to create submodels. You can set the submodeler when you create an Improv instance, through the `submodeler` option. By default, the submodeler is just a function that returns an empty object, but you can set a custom one to perform housekeeping tasks related to setting up the submodel. For example, maybe you want to copy the overarching model's tags to the submodel.

The submodeler takes two arguments: The model that is receiving a submodel, and the name of the submodel being created (as given in the templating directive). It should return a plain object that will be attached to the model as a submodel.

Note that the submodeler is only called if a given submodel doesn't exist already.

CHAPTER 7

Glossary

generator An Improv object created by *`Improv()`*

group An object consisting of two properties, `tags` and `phrases`. `phrases` is just a list of potential phrases the group's parent snippet might produce. `tags` is a list of tags associated with this group, for use in filtering.

phrase An individual section of text (a string) that can be produced by a given snippet, included in one of the snippet's groups.

snippet A specific class of texts intended to be fulfill similar textual role. Each property of a generator's `spec` object is treated as a snippet, consisting of an object with a `groups` property that is itself an array of groups.

tag In Improv, tags are hierarchical markers; the "world model" is defined in terms of a list of tags. Each individual tag is an array of strings, arranged as an increasingly specific list of categories. The `tags` property of models and groups is itself an array of arrays.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index