Imaginarium tutorial

Ian Horswill, 2/22/20, last updated 4/9/21

Contents

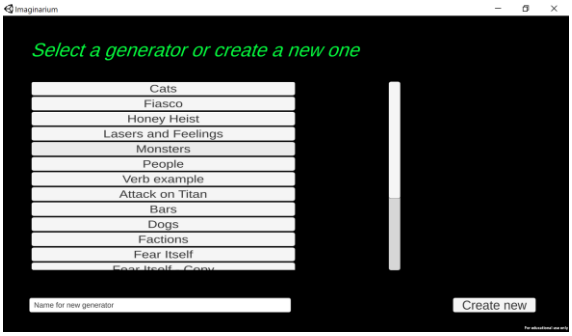
Imagining cats

Imaginarium is a tool that generates random fictitious entities (items, characters, monsters, *etc*.) for use in tabletop role playing. It lets you define programs called *generators* that make random instances of different kinds of objects.

For example, start *Imaginarium*:

Then hit the ESC key to go to the main menu:

Choose “Select generator”, and you get a list of generators on your machine:



And now choose the generator called “Cats.”

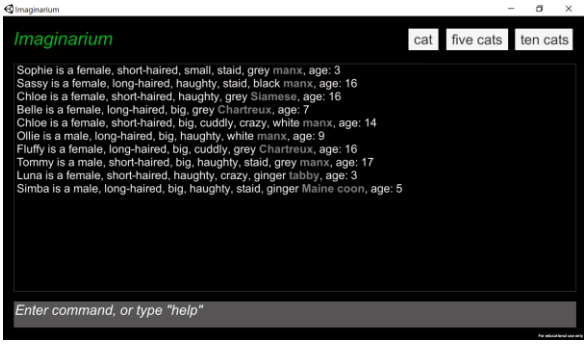
Now type “imagine a cat” at the bottom and hit return, or just press the button labeled “cat”:

and it will print something like:

Each time you hit the return key, it will generate new, random cats:

> Stella is a female, long-haired, big, haughty, staid, tabby, age 16.
> Jasper is a male, short-haired, cuddly, crazy, black manx, age 4.
> Gizmo is a male, long-haired, cuddly, ginger Maine Coon, age 10.

You can also say “imagine ten cats”, and it will show you five at a time. Alternatively, you can just hit the “ten cats” button:



You can also say “imagine 30 big tabbies” and it will give you a whole lot of cats, all of whom happen to be big tabbies, but who all have different names, personalities, colors, etc.

Teaching it new things

Imaginarium is programmable in the sense that you can tell it about new kinds of things and it will generate them for you. You teach it new things using simple English declarative sentences like “`Tabbies are a kind of cat.`”

Or, to be more honest, you program it using a programming language that is superficially similar to English, but is quite limited and lacking in nearly any of the subtleties of human language.

A given kind of entity will be defined by some set of attributes that it can have. The system has been told, for example, that cats have a name, an age, a breed, and can be female, male, long haired, short haired, big, small, haughty, staid, white, black, ginger, etc. Inventing a cat involves choosing a name, age, and breed for the cat, and deciding which of the above adjectives apply to it. However, some combinations of attributes don't make sense. *Imaginarium* knows that a cat can't be simultaneously big and small, for example. These restrictions on possible combinations of attributes are called *constraints*. Imaginarium is a "constraint solver" or "constraint-based programming language", meaning that you tell it about attributes and constraints and then it finds combinations of attributes that satisfy the constraints.

Teaching *Imaginarium* about a new kind of entity involves telling it about its possible attributes and the constraints that apply to them. Adding attributes expands the space of possible objects, while adding constraints restricts it. For example, if we wanted to teach the system about magic users, we could say:

```
A magic user is dark or light
```

That tells the system that:

- There's a kind of entity called a magic user
- They have dark and light as possible attributes
- They have to be one or the other
- They can't be both

If we then say "imagine a magic user", we'll get a pretty boring output:

```
The magic user is a dark magic user
The magic user is a light magic user
```

The system doesn't actually understand English, or much of anything about the world you haven't told it. In this case, it understands that magic users are a kind of entity, but it doesn't know that they're a kind of creature, or character, much less that they would normally have names. For all the system knows, a magic user is a kind of furniture.[1] So when it refers to the object, it doesn't have any better way of doing it than saying "the magic user".

Moreover, the only thing we've told it about magic users besides that they exist is that they can be dark and light but not both. So given what we've told it, there are only two possible magic users: a dark one and a light one. There are no other possible magic users because there are no other possible combinations of attributes for them:

Possible magic users	
Dark	✓
Light	✓

Let's change that by giving it some more possible attributes. We'll tell it about different kinds of magic users:

```
Thaumaturge, necromancer, neopagan, technopagan, and shaman are kinds of
magic user.
```

[1] It should be pointed out that the system doesn't know anything about furniture either, unless you tell it about it.

When you tell the system that a kind of object (magic users) comes in a variety of subkinds (necromancer, etc.), it will ensure that any particular object it makes of that kind is always of one particular subkind. Now we get slightly different results when we hit return:

```
magic user 0 is a dark technopagan
magic user 0 is a light shaman
magic user 0 is a dark necromancer
magic user 0 is a light necromancer
```

The system can generate more varied results because we've given it more to work with. Instead of magic users only involving a dark/light choice, there's now also a choice of type: thaumaturge, necromancer, etc. We've told it about 5 types as well as the two dark/light varieties, so there are 10 possible magic users:

Possible magic users					
	Thaumaturge	Necromancer	Neopagan	Technopagan	Shaman
Dark	✓	✓	✓	✓	✓
Light	✓	✓	✓	✓	✓

The problem is that in many story worlds, there's no such thing as a light necromancer. So we might want to outlaw that combination by adding the constraints:

```
Necromancers are dark
Thaumaturges are light
```

Now we've carved some bits out of the possibility space:

Possible magic users					
	Thaumaturge	Necromancer	Neopagan	Technopagan	Shaman
Dark		✓	✓	✓	✓
Light	✓		✓	✓	✓

The system will guarantee not to generate combinations that violate the constraints you give it.

By the way, if you haven't tried it already, type: `imagine a magic user cat`.

How the system thinks about the world

Up to this point, I've used the deliberately vague language of "entities" and "attributes," treating both the *kind* of thing an entity is (magic user, cat, tabby) and the adjectives (dark, light, short haired) that apply to it with the same nebulous term "attribute." However, the system is designed to interact in pseudo-English, and English makes distinctions between that concepts that are represented through nouns, verbs, and adjectives. *Imaginarium* adopts this distinction, although its understanding of English is very limited. Most of the concepts it understands fall into:

- **Proper nouns** (Fred, The Umbrella Corporation) name *specific entities*.
- **Common nouns** (cat, magic user, Siamese) name a *kind* of entity.
- **Adjectives** (big, small, dark, light) express *yes/no properties* of entities.
- **Verbs** (to love, to go to school at) express *relationships* between pairs of entities

There are other kinds of concepts it understands, but we'll get to those later.

When you tell the system about a new concept, it remembers whether you used it as a noun, verb, or adjective, and will always use it as such in the future. If you say "A glorp is a kind of monster", it will know that glorp and monster are both nouns and that in future when telling you that an entity has the glorp attribute, it should tell you that by calling it, e.g. "a glorp", rather than "a glorpish monster" or "an entity that's glorp". It also knows that once it says the entity is a glorp, it doesn't also need to tell you it's a monster, because that's implied by it being a glorp.

How to tell the system about your favorite stuff

You program the system by typing statements that are true about whatever kinds of entities you want it to generate. However, the system only understands specific kinds of statements that are written in specific sentences patterns. Here's a basic introduction to the kinds of information you can tell it about and what sentence patterns to use.

Kinds

Kinds of entities are expressed as common nouns like `cat`, book, `chair`, or `candy factory`. Notice from the latter that *Imaginarium* is okay with a "noun" consisting of a series of words. That lets you say things like `candy factory` without having to teach it separately about what `candy` means, irrespective of factories, and `factory`, irrespective of candy.

For the most part, you can teach the system a new kind (a new noun) by just using it in a statement someplace. For example, if you say:

```
Chairs are wooden, steel, or plastic.
```

Then it will figure out that `chair` is a noun, even if you haven't previously used the word `chair`.[2] Kinds are important because the other kinds of words tend to get organized around them. For example, you don't tell it "fluffy is an adjective", you tell it:

```
Cats can be fluffy.
```

Which tells it that it needs to worry about entities being fluffy when those entities are cats. But if you told it about some other kind of entity, like books, it doesn't need to worry about whether they're fluffy unless for some reason you also specifically told it: `books can be fluffy`.

Nouns can have multiple words

Imaginarium allows nouns to have more than one word, such as in the example `candy factory`, above.[3] This is really useful, but it's also a little complicated. How does the system know that `candy factory` is a two-word noun, but `black cat` is a one-word noun with an adjective attached? The answer is that it always assumes something is a multiword noun, unless you've already taught it that the words inside the noun have other meanings. So if you say:

[2] And, similarly, it will learn that `wooden`, `steel`, and `plastic` are adjectives.

[3] Linguists call these "phrasal nouns." They're phrases, but they act like single nouns. For example, a "face-off" is not an off with the extra property of being face such that there can be offs that are not face. Face-off acts as if it were a single word, even though it's spelled as two.

```
A cat can be black.
Black cats are great.
```

Then the first line teaches it that cat and black are separate concepts, and the second that when you have a cat who is specifically black then that cat will always be great. But if you just say the second sentence, without ever having used the words `black` or `cat`, then it will assume that `black cat` is a single, indivisible noun and subsequent uses of `cat`, without `black`, will confuse it.

Subkinds and superkinds

You can teach the system that one kind is a subkind of another kind by saying:

noun `is a kind of` *noun*`.`

For example, "`chair is a kind of furniture`" or "`sword is a kind of weapon`". This tells it that all `swords` are also `weapons`. This can get tedious when you have to teach it about a lot of kinds, so you can just say:

```
Sword, gun, and club are kinds of weapon.
```

And it will treat it as equivalent to typing three separate "is a kind of" statements. Note that you have to use the commas here so that it doesn't think there's a kind of weapon called a "sword gun", although if you want that, take out the comma.

Note that when you tell the system to generate an entity of some kind that has subkinds, it will always choose a specific subkind. That is, if you say "`imagine a weapon`", it will always generate a `sword`, gun, or `club`, not some generic thing described only as "`a weapon`". It will generate the different subkinds with equal probability. But if you want one to be more common than the others, you can put how much more common in parentheses:

```
Sword (2), gun, and club are kinds of weapon.
```

This says that swords are twice as common as guns and clubs in our world. If you want guns to be rare, you can put in a fractional number:

```
Sword, gun (0.1), and club are kinds of weapon.
```

Adjectives

As with nouns, you can tell the system about an adjective just by using it. Also like nouns, adjectives can consist of short phrases rather than single words. You generally teach the system about a new adjective by saying one of:

Noun `is/are` *adjective*`.`
Noun `is/are` *adjective*`, …, or` *adjective*`.`
Noun `can be` *adjective*`.`
Noun `can be` *adjective*`, …, or` *adjective*`.`

These all introduce relationships between *noun*s and *adjective*s but have slightly different meanings. When you say "or" you're saying the different *adjective*s are alternatives, meaning the *noun* can't be more than one of them at a time. When you say `is/are`, you're saying the *noun* always has that property (or one of the alternatives). When you say `can be`, you're saying the property is optional. It might be true but doesn't have to be. For example, if you say:

```
A chair is solid.
```

You're saying all chairs are always solid. But if you say:

```
A chair can be comfy.
```

You're saying some chairs are comfy chairs and others aren't.

If you say:

```
Chairs are sturdy or cheaply built.
```

You're saying all chairs are either study chairs or cheaply built chairs, but never both, i.e. there are no sturdy cheaply build chairs.

But if you say:

```
Chairs can be sturdy or cheaply built.
```

You're saying chairs can be study, cheaply built, or neither (but still not both). Thus, there are sturdy chairs, cheaply built chairs, and (just plain old) chairs that are neither.

Relationships

Relationships between pairs of entities are expressed by verbs. You can teach it about new relationships by telling it what kinds of entities the verb relates:

Noun can *verb* some *noun*

You can put whatever *noun*s and *verb* you like, and again, the verb can consist of multiple words, such as:

```
People can invest in some stocks.
```

This tells the system there's a kind of entity, `person`, and another kind of entity, `stock`. Moreover, people and stocks are related by the relation "`invest in`". So there's some set of people, and some set of stocks, and any given person may or may not be investing in that stock. This leaves open the possibility for a given person to invest in many stocks or no stocks. And similarly, for a stock to have many or no people investing in it. So when you ask the system to imagine people and stocks, it will feel free to structure who's investing in what (or not) completely randomly.

By contrast, if we change `some` to `one`:

```
People can invest in one stock.
```

It tells the system that people can't invest in more than one stock. And if we change `can` to `must`:

```
People must invest in one stock.
```

Then the system will make sure all people invest in exactly one stock. Here's a summary of what the different sentence patterns mean:

Sentence pattern	**Meaning**
Subject `can` *Verb* `some` *Objects*	Any *subject* can *verb* any *object*, but doesn't have to.
Subject `must` *Verb* `some` *Objects*	Every *subject* must *verb* at least one *object*.
Subject `can` *Verb* `one` *Object*	Each *subject* can *verb* up to one *object*.
Subject `must` *Verb* `one` *Object*	Each *subject* must *verb* exactly one *object*.

You can also say things like "`people must invest in two stocks`" or "`up to three stocks`". "`at least four stocks`", etc.

Relationships within a kind

The subject- and object-kinds of a verb are often the same:

```
People can know many people.
```

This is a common case and *Imaginarium* understands a number of common special cases of these relations. If you say:

```
People can know many people.
```

The system understands that know is a relation between people. But if we say:

```
People can know each other.
```

The system still understands that know is a relation between people, but it also understands that if I know you, then you also know me.

If we say:

```
A person can be friends with some people.
```

Then we leave open the possibility of a person being friends with themselves, which may or may not match your intent. But if you say:

```
A person can be friends with other people.
```

Then we rule out people being friends with themselves. We could also achieve the same result by adding:

```
People can't be friends with themselves.
```

Alternatively, we can force people be friends with themselves by saying:

```
People always are friends with themselves.
```

One last thing to point out: as far as the system is concerned here, the verb is "`be friends with,`" not just "`be friends`". That is, it considers the preposition to be part of the verb. It's not optional, it can't be moved around, and it can't be combined with other prepositions, because *Imaginarium* doesn't know what a preposition is in the first place.

Kinds of relationships

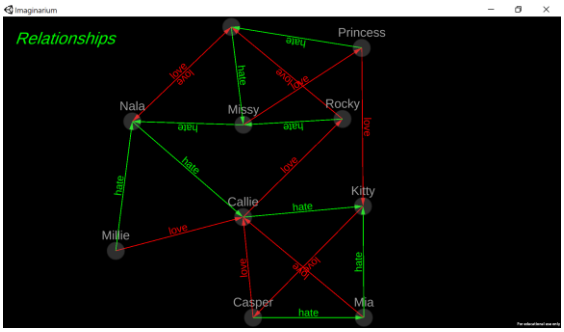
Just as you can say that one noun is a special case of another noun by using the "`is a kind of`" construction, you can tell the system that one verb is a special case of another using the "`is a way of`" construction:

```
Being friends with is a way of knowing
Being coworkers with is a way of knowing
```

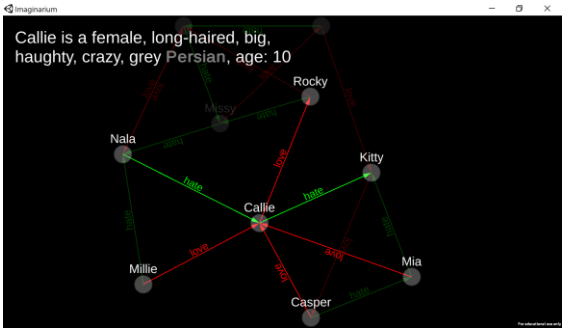
This tells the system that if A is friends with B or is coworkers with B, then A also knows B. As with the "kind of" construction, if a verb has special cases like this, then the system will make sure that exactly one special case applies. In this case, if two characters know each other, they must also be friends with or coworkers with one another, but they cannot be both.

Visualizing relationships

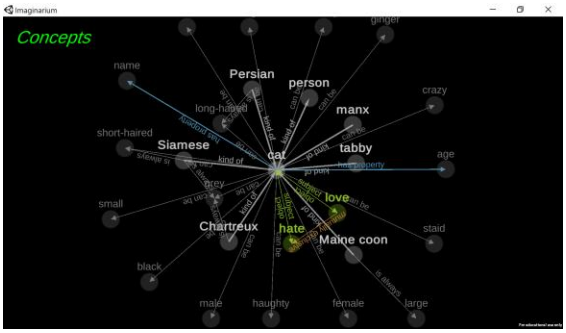
Assuming you're still using the cat generator, press the "ten cats" button. This will make 10 cats. When it prints the text out, it doesn't print relationships because there are usually too many of them and the output gets cluttered. But press the TAB key. This will bring you to the Relationships visualization:



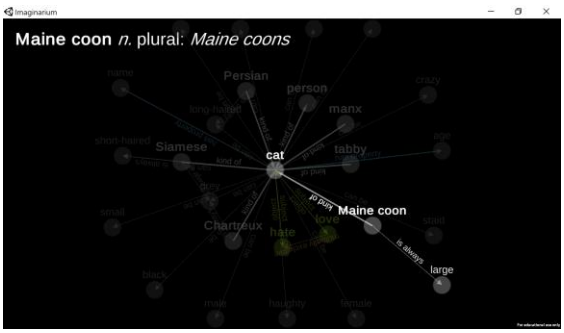
This lets you see all the cats and who loves or hates whom. If you mouse over one of the cats, it will also show you the information about that particular cat:



If you hit TAB again, you get a different display, the Concepts display. This shows the relationships between all the concepts (noun, verbs, adjectives, etc.) that you've defined:



Once again, if you mouse over a specific concept, it will display information about it:



And, finally, if you hit TAB one more time, it will return you to the main screen. You can cycle between these screens as much as you want by just hitting tab.

Implications

There are a number of other kinds of things you can tell *Imaginarium*. Probably the most important of these are various forms of implication. For example, you can tell it that certain adjectives always apply to certain kinds of entities:

```
Persians are long-haired
Main coons are large and long-haired
```

You can also state implications for verbs. If you say:

```
People can know each other
People can be friends with each other
People can love other people
Being friends with implies knowing
Loving implies knowing
```

You're saying that knowing is a symmetric relation (if A knows B, B know A), and so is friendship. Moreover, you can't be friends with someone without knowing them. But love is a relation that isn't necessarily symmetric: A can love B, without B loving A. But love still implies knowing. So if A loves B, then A knows B, and so B knows A, but B may or may not love A ☹.

Modifiers on nouns

You can attach modifiers (adjectives and other nouns) to nouns in most of the places you can use nouns. For example:

```
Angry cats are dangerous
```

Tells the system that cats are dangerous when they are angry, but doesn't say that non-angry cats are dangerous. On the other hand, it doesn't say they have to be non-dangerous either. Right now there's no construction like "cats are dangerous only when they're angry". But you could achieve the same end in this case, but adding:

```
Dangerous cats are angry
```

These two statements together would specify that cats are dangerous if and only if they're angry.

Other kinds of attributes

So far, the only kinds of attributes we've attached to entities are adjectives and relationships. But there are a few other things you can attach. One is numbers:

```
A character has an age between 20 and 60
```

Tells the system characters have a property called age, that's a number, and that it's in the range 20-60 (inclusive).

You can also give characters names. For example, if you add a file to your project called "cat names.txt" and put a list of names in it, then you can say:

```
Cats have a name from the list cat names
```

You can use this to attach any textual attribute you like to a character, so long as you put the list of possible values in a file. If the attribute happens to be called "name", then the system will know to refer to the cat by their name rather than just calling them "cat 0".

A more complicated example is:

```
Humans have a given name from the list given names
Humans have a surname from the list surnames
A human is identified as "[given name] [surname]".
```

The first two of these tell the system that humans have both a given name and a surname, and from which lists to draw them. The second tells the system that when referring to humans, rather than calling them "human 0", it should refer to them by their given name, followed by their surname.

Parts

You can tell the system that entities can have other entities as parts. For example if you say:

```
A bear has a hat called their favorite hat.
```

Then the system will know that when it makes a bear entity, it also needs to make a hat entity, and that they're linked by the favorite hat relationship. You can also say things like "a bear has two hats called their favorite hats" and it will make two separate favorite hat parts. And you can omit the "called …" part if you like.

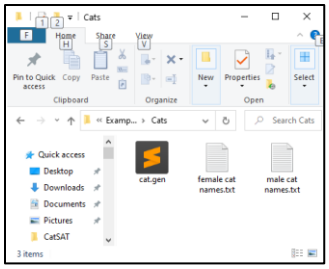
Please note that Parts are limited at the moment. In particular, the system needs to know in advance what parts there will be. So if you say something like:

```
Orcs and balrogs are kinds of monster
Orcs have a film called their secret guilty pleasure
```

Then you can say "`imagine an orc`" and it will make both an orc and their secret guilty pleasure film, or you can say "`imagine a balrog`" and it will make just the balrog because they don't have a secret guilty pleasure film. But if you say "`imagine a monster`" and it generates an orc, the orc won't have a secret guilty pleasure film. That will be fixed sometime in the future.

Navigating the folder structure

So far, we've glossed over the fact that generators are stored in files. Hit ESC again to go to the main menu and press `Show Generator Folder`. On Windows, you should see something like this:



You'll probably see a different icon for the file `cat.gen` because your operating system probably doesn't know what application to use with `.gen` files. The system also may not display .txt in the names of the files, depending on the settings on your computer.

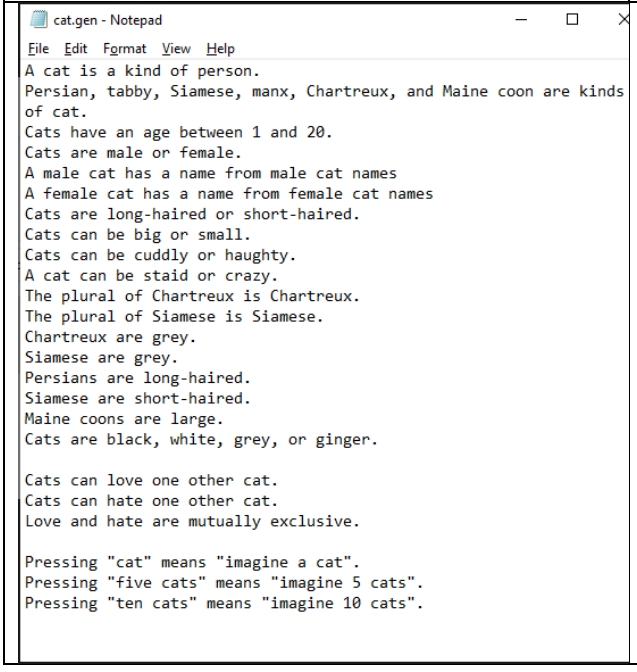
Each generator is stored on disk as its own folder. Inside the folder are .gen files, and optionally text (`.txt`) files, which where you store names and other lists for properties.

The commands for your generator are stored in `.gen` files. They're just normal text files that happen to have commands like the ones above stored in them, one per line. You can have as many .gen files as you like, but you need at least one. However, we recommend you name the files after the kind of noun they're defining. The `cat.gen` file is full of cats, so it's called `cat.gen`. If you added dogs into the mix, you might have a `dog.gen` file. Or you could just put them all in one file. That's fine too.

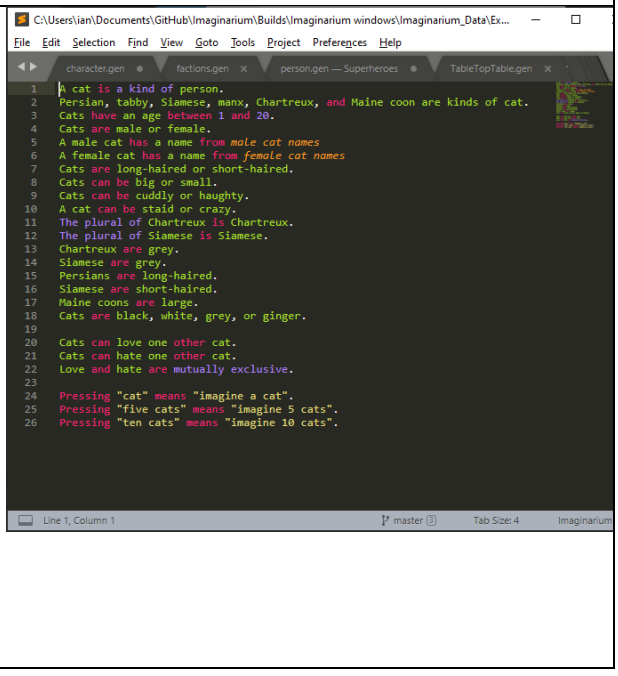
Open up the `cat.gen` file. If you're doing this for the first time, the operating system will probably ask you to choose what application to use to edit it. If you already have a good code editor installed on your system, like Visual Studio, Sublime, or Emacs, we recommend you use that. If you don't, or don't know what a code editor is, we recommend you install Visual Studio Code, since it has some special support for editing *Imaginarum* files.

When you open up the cat.gen file, you should see something like this:

Notepad



Sublime text



If you're using Sublime text, you can use the "Configure sublime text" button in the main menu to turn on syntax highlighting as is shown in the screenshot above. If you're using Visual Studio Code, you can go to the Extensions manager (click the icon on the left that looks like little blocks) and type "imaginarium" in the search bar; then tell it to install the extension that comes up.

Now try editing the cat.gen file:

- Change the text on screen. For example add a line "`Burmese is a kind of cat`". Each command must be on its own line.
- Save the file. *Imaginarium* won't know anything has changed if you don't save it.
- Now go back to *Imaginarium* and hit ESC to go in to the main menu, and then hit it again to go back to the main screen. This will cause it to reload the generator from your files.
- Now type "`imagine 10 Burmese`". You should see a bunch of random Burmese cats.

You can keep changing any of the files in the generator folder as you like. Each time, just be sure to save your files and hit ESC twice.

You may get tired of typing "`imagine a` …" over and over again. That's why you can use the:

`Pressing "X" means "imagine Y"`

command, that you see at the end of the file. You can add your own lines to make your own buttons on the screen. For example:

`Pressing "my button" means "imagine a Burmese"`

will add a button to the screen that says "my button" and pressing it will run the command "`imagine a Burmese`".

We recommend you look around at the `.gen` files for the different example generators included with *Imaginarium*.

Making a brand new generator

To make a new generator, you need to make a new folder for it. There are a few ways of doing this. The most basic way is to:

- Go into the `Select Generator` screen
- Type the name for your new generator at the bottom
- Press `Create New`
- Hit ESC to go the main menu and then press `Show Generator Folder`

Now you can add files to your heart's content. A common thing to do is to start by stealing files from other generators. You can then edit them as you like.

Alternatively, you could start from an existing generator as a base:

- Go to an existing generator (e.g. using `Show Generator Folder`)
- To the parent folder of the generator
- In the folder window, make a copy of the folder for the old generator
- Then, back in *Imaginarium*, go to the `Select Generator` screen and choose it.
- You're ready to start editing it as you like!

One last point: Imaginarium only looks in a few folders for generators:

- The `Examples` folder inside the application
- Inside the Documents folder, there is an Imaginarium folder, and inside that is a Generators folder. It will search the Generators folder for folders holding generators.

In general, you should put your generator folders inside the second of these (the folder called Generators) inside the Imaginarium folder of your Documents folder).

Advanced features

Automated testing

When you make complex generators, it's possible to introduce bugs. They could just be typos or they could involve your constraints being over-broad, ruling out thing you didn't intend to rule out, or over-narrow, allowing things you didn't intend to allow. That means you need to test the system out by checking that the things that ought to exist do (i.e. it can indeed generate the kinds of things it ought to be able to generate), and that the things that ought not to exist don't (it can't generate things that it shouldn't be able to).

So in practice, you don't just write your code, you try it out – you kick the tires on it and generally check that it seems to be behaving the way you'd expect. When you find something wrong, you change the code. But changing the code may or may not fix the problem, and it might also introduce a new bug. So if you really want things to be perfect, you have to go back and test all over again. This is just the nature of software development. People all around the world spend their days fixing and testing.

But it's a pain to have to keep retyping the same tests over and over again. And in addition, you might forget one, mistype it, or otherwise introduce new errors in the testing itself. So in practice, software designers tell the machine the tests to perform but then have the machine run the tests itself and just report the results. This is called automated testing.

You can automate testing of your generators by adding tests to your `.gen` files. Imaginarium will remember them, but otherwise ignore them until you give the command "`test`", at which point it will run all the tests you've specified. Tests are of the forms:

- *Noun phrase* `should not exist`
- *Noun phrase* `should exist`
- `Every kind of` *noun phrase* `should exist`

Where *noun phrase* is either a noun or a noun with some adjectives. For example:

```
Short haired Persians should not exist
```

When you run the `test` command, *Imaginarium* finds all the tests you give it and tries to `imagine` each of their *noun phrase*s, looking to see if it can successfully generate instances of them. If it can generate one that's labeled `should not exist`, it displays an error message and gives you the instance it generated. Alternatively if it can't generate one that was labeled `should exist`, then it prints an error message about that.

The "`every kind of`" test is the most useful. It tells the system to find all the different subkinds, subsubkinds, etc., of the noun you specify and test them individually. So if you say:

```
Every kind of haughty cat should exist
```

It will go through every individual breed of cat and test to make sure it can imagine a haughty one.

Git support

If you are a user of git, you can use *Imaginarium* to clone git repos. That way, you can make a repo containing several generators (each should be a folder at the top level of the repo) and clone/update it through *Imaginarium*'s Repo Manager. Please note that the git library we're using is limited to cloning and completely replacing the contents of a folder with current head on your server. It doesn't do pushes or merges. If you want to push or merge, you will need to do that using a real git client. But this feature works well enough for easily sharing a group of generators with your friends.

Using *Imaginarium* in digital games

Finally, Imaginarium is available without the user interface elements as a drop-in DLL. If you are developing a digital game under Unity, or some other platform that supports C#/.NET/Mono, you can import your *Imaginarium* generator into your game by adding the DLL and your generator directory to your game. See https://github.com/ianhorswill/ImaginariumCore for more information.