Random Generator

A random generator for pretty much anything. *By Nicholas Paul*

The Random Generator can be used to generate things like secure passwords, awesome dungeon loot, fake names, and even the basics like numbers and letters. The random generator works by picking an item randomly from a list defined in a text file. Check out the syntax or take a look at the examples below.

Contents

- Syntax
  - Lists
    - Format
    - Anonymous Lists
    - Special Lists
    - Ranges
    - Include
  - Comments
  - Escape Characters
  - Tags
    - Special Tags
    - Percent
    - Plural
  - Binding
    - List Name Binding
    - String Literal Binding
  - Utilities
    - Capitalization
    - A and An
    - Repeat
    - Plurality
    - Replace
- Examples
  - RPG Name
  - Spoken Number
  - Password
  - Pronounceable Password
  - Hex Color
  - Card
  - Recursion

Syntax

Lists

Lists must be contained in .txt files and can be referenced at any time by using an include in the editor and running the script.

Format

List names must be one word consisting only of alphanumeric characters and underscores. **List items** must each be entered on a separate line below the list name. Empty lines are ignored and the list won't end until another list name or the end of the file is reached. Lists are referenced using square brackets.

```
#color1
brown
night black
orange

#colored_fox
[color1] fox
red fox
tan fox
```

NOTE: If a referenced list has an invalid name, a name that doesn't belong to a list, or is empty, it will be treated as a normal string.

```
[colored_wolf]
[]
[%this]
```

Anonymous Lists

Like list name references, anonymous lists are denoted using the square brackets. All items in the list must be separated using the vertical bar '|'. Anonymous lists can contain empty strings.

```
My favorite color is [red|green|blue].
The list [cat|dog||fox] will return either "cat", "dog", "fox", or nothing at all.
```

Items in a list will be parsed *after* being chosen. Thus enabling nesting lists within lists.

```
Edward is [[nice_adj]|[mean_adj]|[odd|cool]]
```

Special Lists

Ranges

Ranges will generate a random number or lowercase character based on the range using the syntax below.

```
I saw [5-10] apples fall from the tree.
The license plate read [a-z][0-9][a-z][0-9][g-q][0-9].
```

Include

The list named INCLUDE is reserved for importing files. All files in the list will be imported in the order that they appear. All file paths are evaluated relative to the file containing the include tag. For example if you included a file on your desktop using `C:\Users\Name\Desktop\data.txt` and you wanted to use this file to include several other files on your desktop, data.txt might look like:

```
#INCLUDE
data2.txt
data3.txt
C:\Users\Name\Desktop\data4.txt //Not necessary but still works
```

NOTE: Beware of escape characters! In the filepath `C:\Users\Name\Desktop\temp.txt` the \t in `Desktop\temp.txt` will produce a tab. To fix this use a forward slash: `C:/Users/Name/Desktop/temp.txt`

Comments

Comments can be used anywhere in the file to give readers brief descriptions of your list, an intro to the file, or anything else. Here are some ways they can be used:

```
//The tool lists contains basic handyman tools
#tool
//carpentry items
saw //Comments can be used on the same line as items
//other items
drill
hammer //This will not produce a space after hammer
```

Escape Characters

Escape characters are used to bypass certian special characters. For example, if you wanted to create an item in a list that starts with the # character you may write

```
#listname
#item1
```

This will not create a list named `listname` with an item named `#item1`, but two separate empty lists. This is because the parser thinks that you are defining a new list called `item1`. To bypass this, use the escape character '\'

```
#listname
\#item1
```

Valid escape characters:

- `\#` Escapes list definitions
- `\//` Escapes comments. An item named `Hammer \//a hammer` will produce `Hammer //a hammer`
- `\{` and `\}` Escape tag blocks
- `\!` The ! character is usually used for binding and utilities `\!Abc(test)` will return the string as-is: `!Abc(test)`
- `\[` and `\]` Escape the end or beginning of a list reference. `[\[|\]]` will return either a [or a]
- `\|` Escape anonymous list item breaks. `[\||I]` will return a | or a I
- `\,` Escape utility arguments. `!x(5,[0-9],\,)` will produce a list of numbers something like `0,0,7,4,8`
- `\n` Parses to a new line character `!x(5,[0-9],\n)` will produce 5 numbers each on their own line.
- `\t` Parses to a tab character

Tags

Additional data can be stored along with an item by enclosing the data as a comma seperated list in curly braces. The data can then be called using a '.' in the list name. See the examples below.

```
#sport
baseball {ball:baseball, other:bat}
ultimate frisbee {ball:frisbee, other:none}

[in :] The sport I play uses a [sport.ball].
[out:] The sport I play uses a frisbee.
```

Default values are specified using the optional 'or' keyword.

```
#utensil
pen {color:red}
pencil

[in :] The utensil draws a [utensil.color or black] line.
[out:] The utensil draws a black line.
```

Tags will be parsed so they can contain more then just plain text.

```
#warm_color
red
yellow

#cold_color
blue
purple

#color
[[warm_color]|[cold_color]] {warm:[warm_color], cold:[cold_color]}

[in :] [Color.warm] is a warm color, [color.cold] is a cold color, and [color] may
be either warm or cold.
```

Tags do not work in anonymous lists. The data is stored, but there is no way to access it. The only exception to this rule is the percent tag and the plural tag.

```
#double
thing {tag1:[one {q:q,p:p}|two {q:u,p:v}], tag2:two}
//Calling [double.tag1.q] will not give an error, but will produce unwanted results

//The 70% tag can still be used
The card was a [[1-10]|[A|J|K|Q]{%:70}][♥|♦|♣|♠].
```

Special Tags

Percent

The percent tag is always named the % character. It indicates the percentage chance that the item will be added to the random selection list when an item is requested. This tag **cannot** be called by using [listname.%]. Unlike any other tag, table this tag can be used in anonymous lists.

```
#treasure chest
[10-50] gold coins
[100-300] gold coins {%:50}
[1000-2000] gold coins {%:10}
```

The list

```
#b_or_c
b {%:10}
c {%:10}

[in :] the letter is a [b_or_c].
[out:] the letter is a . //But will give the warning "The list 'b_or_c' contains no
elements"
```

Plurality Tag

This tag is a special tag that is primarily used with the !s() utility. It can also be called normally.

```
#animal
bear
fox {s:foxes}

[in :] I saw a few [animal.s or [animal]s] on our trip up north.
```

Binding

List Name Binding

```
#tool
drill {use:drill screws}
hammer {use:pound nails}
tape {use:attach things}
axe

[in :] I use a [tool = tool1] to !tool1(use).
[out:] I use a hammer to pound nails.

//You can also use default values
[in :] I use a [tool = tool1] to !tool1(use,...actually\, I can't remember).
[out:] I use a axe to ...actually, I can't remember.
```

String Literal Binding

`!let(`*`name`*`,`*`string`*`)`

The Utility !let will evaluate `string` and then bind the evaluated expression as a literal string to `name`. The string can then be accessed by calling `!name`. Variable names must be alphanumeric and cannot be the same as any utilities. Calling `let` on the same variable name more then once will simply overwrite the binding attached to that variable.

```
[in :] !let(favSport,[sport])
[out:] soccer

[in :] My favorite sport is !favSport.
[out:] My favorite sport is soccer.

//The second argument can contain anything
[in :] !let(favSport,[Sport] is my favorite[\!|.])  //Bindings can be overwritten
[out:] Baseball is my favorite!
```

Utilities

Capitalization

`!Abc(`*`expression`*`)`

Formats `expression` using various capitalization rules

Title capitalization Makes the first letter of every word uppercase.

```
[in :] !Abc([guy] and [girl])
[out:] Jeff And Katie
```

All Uppercase Makes all letters uppercase.

```
[in :] !ABC([shout]\!)\!
[out:] AAAHHH!!
```

All lowercase Makes all letters lowercase.

```
[in :] !abc(!ABC([laugh]))
[out:] hahaha
```

An and A

`!an(`*`expression`*`)`

Parses `expresion` and decides whether or not to use 'a' or 'an' before the phrase and return the decision, a space, and the evaluated expression.
If 'a' or 'an' needs to be capitalized, simply capitalize the utility name: `!An(`*`expression`*`)`. The decision is based off of the rules listed on Purdue Owl.

```
[in :] !an([animal])
[out:] a unicorn

[in :] !An(honorable man)
[out:] An honorable man
```

Repeat

`!x(`*`low`*`{-`*`high`*`},`*`repeated_expression`*`{,`*`spacer_expression`*`})`

Repeats `repeated_expression` a random number (defined by `low` and `high` (optional)) of times and puts `spacer_expression` (also optional) between the repeats.
Anything in {braces} in the syntax guide can be omitted.

```
[in :] !x(2,ha)
```

```
[out:] haha

[in :] !x(4,[do|de|da],-)
[out:] do-de-da-de

[in :] I like !x(2-4,[color],\, ) and [color].
[out:] I like orange, black, yellow and pink.
```

NOTE: `repeated_expression` and `spacer_expression` will be repeated *as is* and then parsed after the fact. For example,

```
Input:
I like !x(2,[fruit], and ).

After evaluating repeat:
I like [fruit] and [fruit].

Output:
I like apple and orange.
```

Plurality

`!s(`*`amount`*`,`*`list`*`)`

Will make the item pulled from `list` plural by analyzing `amount`.
`amount` is some string representation of an amount and `list` is the name of a list (without brackets) to be evaluated.
`amount` will be parsed before being evaluated so lists, ranges, and literals can be included.

```
!s([1-5],fruit)
!s(two,color)
!s([number],animal) //Where number is a list called "number"
```

Note: By default, an s is added to the end of the list item so *apple* will become *apples* and *dog* will become *dogs*. There are some instances where adding an s to the end of the word doesn't make sense. In those cases, a plural form can be defined as a plurality tag in the list.

```
#animal
dog
fox {s:foxes}
cat
```

Replace

`!replace(`*`replace_this`*`,`*`with_this`*`,`*`in_this`*`)`

Replaces every instance of `replace_this` in `in_this` with `with_this`

```
[in :] !replace( ,_,Hello World)
[out:] Hello_World

//It will make the replacements before parsing
[in :] !replace([,_,[color])
[out:] _color]

[in :] !replace([color],[sport],My favorite color is [color])
[out:] My favorite color is soccer
```

Examples

Below are some basic examples of random generators. More examples are included in the `lists` folder of the project download.

RPG Name

Shortened version of a generator for RPG-like names

```
#first_part
grim
gor
ano
hen
imp
nim
jac
kin
lob
mon
nop
oxen
pers
wad
win
rad

#middle_part
[a|e|i|o][b|c|d|f|g|h|j|k|l|m|p|r|s|t|v|w][|[middle_part]{%:20}]

#last_part
ah
ly
mia
fal
el
ex
ent
or
lorg
rod
#rpg_name
[first_part][middle_part]
```

```
[first_part][middle_part][last_part]
[first_part][last_part]
```

Sample Output

```
Gorel
Henly
Radod
Anofah
```

Spoken Number Generator

Generates a random, written out, number from one to one million.

Script

```
#one
one
two
three
four
five
six
seven
eight
nine

#ten
[ten|eleven|twelve|thirteen|fourteen|fifteen|sixteen|seventeen|eighteen|nineteen]
twenty[{%:10}|\_[one]]
thirty[{%:10}|\_[one]]
forty[{%:10}|\_[one]]
fifty[{%:10}|\_[one]]
sixty[{%:10}|\_[one]]
seventy[{%:10}|\_[one]]
eighty[{%:10}|\_[one]]
ninety[{%:10}|\_[one]]

#hundred
[one] {%:30}
[ten] {%:30}
[one] hundred [ten]

#thousand
[hundred] thousand [hundred]

#million
[hundred] million [thousand]
```

Sample Output

```
two hundred twenty three million four thousand one hundred forty two
```

```
four hundred twenty eight million ninety nine thousand ninety one
nine hundred fifty three million forty nine thousand nine hundred fifty
six million six hundred forty five thousand nine hundred sixty four
```

Password

```
!x(8-12,[[a-z]|!Abc([a-z])|[0-9]|_{%:30}|[@|#|$|%|^|&|*|\!]{%:60}])
```

Pronounceable Password

```
!x(3-5,[b|c|d|f|g|h|j|k|l|m|n|p|r|s|t|v|w|z],[a|e|i|o|u|y{%:50}])[[0-999]|]
```

Hex Color

```
\#!x(6,[!ABC([a-f])|[0-9]])
```

Card

```
[A|2|3|4|5|6|7|8|9|10|J|Q|K][♥|♦|♣|♠].
```

Recursion

```
//Calls itself recursively. There is a small chance that it won't and the line will
end.

#rec
{%:12}
-[rec]

[in :] [rec]
[out:] -------
[out:] ----------
[out:] ---
```