dndice

Release 2.1.2

Oct 07, 2020

Contents:

This is a python package aimed at performing rolls in a syntax that extends that used by D&D. The full list of allowed operators is given by the page *Operator definitions* but most of the time you will use one of a few:

- "1d20" means "roll one 20-sided die." These get evaluated before any of the common operations like addition or multiplication.
- "+", "-", and the other simple arithmetic operators work as expected.
- Parentheses can be used as expected to force some expressions to be evaluated first.
- "2d20h1" means a d20 roll with advantage. More formally, it means "roll two 20-sided dice, then take the highest one." Similarly, "2d20l1" is disadvantage, as it takes the lowest one.
- "1d20r1" means a d20 roll with the halfling's "lucky" trait. It specifically means "roll a 20-sided die, then if the roll is a 1, reroll it and take the new result."

Installing this package through PyPI also installs the script `roll` that allows you to perform rolls from the command line. It provides a variety of switches to explore the full functionality of this package. The source of this script is found at `dndice/roller.py` in the repository. If all you want is a way to roll dice with code, there you go. If you instead want to integrate this with something you're making, read on.

CHAPTER 1

Basic functions

This module provides a few basic functions for interaction with the outside. These should be enough for anyone who just wants to use this library without extending it.

1.1 basic

`dndice.basic`(*expr: Union[str, int, float, dndice.lib.evaltree.EvalTree], mode: dndice.core.Mode = <Mode.NORMAL: 0>, modifiers=0*) → Union[int, float]

Roll an expression and return just the end result.

Parameters

- **`expr`** – The rollable string or precompiled expression tree.
- **`mode`** – Roll this as an average, a critical hit, or to find the maximum value.
- **`modifiers`** – A number that can be added on to the expression at the very end.

Returns The final number that is calculated.

1.2 verbose

`dndice.verbose`(*expr: Union[str, int, float, dndice.lib.evaltree.EvalTree], mode: dndice.core.Mode = <Mode.NORMAL: 0>, modifiers=0*) → str

Create a string that shows the actual values rolled alongside the final value.

Parameters

- **`expr`** – The rollable string or precompiled expression tree.
- **`mode`** – Roll this as an average, a critical hit, or to find the maximum value.
- **`modifiers`** – A number that can be added on to the expression at the very end.

Returns A string showing the expression with rolls evaluated alongside the final result.

1.3 `compile`

`dndice.`**`compile`**(*expr: Union[str, int, float], modifiers=0*) → dndice.lib.evaltree.EvalTree

Parse an expression into an evaluation tree to save time at later executions.

You want to use this when the particular expression is going to be used many times. For instance, for D&D, d20 rolls, possibly with advantage or disadvantage, are used all over. Precompiling those and referencing the compiled versions is therefore very likely to be worth the extra step.

Parameters

- **`expr`** – The rollable string.
- **`modifiers`** – A number that can be added on to the expression at the very end.

Returns An evaluation tree that can be passed to one of the roll functions or be manipulated on its own.

1.4 `Mode`

`class` `dndice.`**`Mode`**

Change the way that a roll is performed.

Average makes each die give back the average value, which ends up with halves for the normal even-sided dice. Critical causes a roll to be like the damage roll of a critical hit, which means roll each die twice as many times. Max makes each die give its maximum value. This isn't really used as far as I can tell.

Higher modes overwrite lower, so MAX supersedes CRIT which supersedes AVERAGE.

CHAPTER 2

Operator definitions

Operator	Expression	Meaning
!	*x*!	Calculate the factorial of *x*.
d	*x***d***y*	Take a *y*-sided die and roll *x* of them. *y* can be an integer, and works just as you would expect. It can also
da	*x***da***y*	Take a *y*-sided die and return the average as if *x* of them had been rolled. This returns an unrounded numb
dc	*x***dc***y*	Roll a critical hit, where the number of dice rolled is doubled.
dm	*x***dm***y*	Roll the maximum on every die rolled.
h	*ROLL***h***n*	After making a roll, discard all but the highest *n* of the rolls. Hint: 2d20h1 is advantage.
l	*ROLL***l***n*	After making a roll, discard all but the lowest *n* of the rolls. Hint: 2d20l1 is disadvantage.
f	*ROLL***f***n*	After making a roll, treat any value that is less than *n* as *n*.
c	*ROLL***c***n*	After making a roll, treat any value that is greater than *n* as *n*.
r	*ROLL***r***n*	After making a roll, look at all of them and reroll any that are equal to *n*, reroll those, and take the result.
R	*ROLL***R***n*	After making a roll, look at all of them and reroll any that are equal to *n* and reroll those. If that number c
r> or rh	*ROLL***rh***n*	After making a roll, look at all of them and reroll any that are strictly greater than *n*, reroll those, and take
R> or Rh	*ROLL***Rh***n*	After making a roll, look at all of them and reroll any that are greater than *n* and reroll those. If a number
r< or rl	*ROLL***rl***n*	After making a roll, look at all of them and reroll any that are strictly less than *n*, reroll those, and take the
R< or Rl	*ROLL***Rl***n*	After making a roll, look at all of them and reroll any that are less than *n* and reroll those. If a number less
t	*ROLL***t***n*	After making the roll, count the number of rolls that were at least *n*.
T	*ROLL***T***n*	After making the roll, count the number of rolls that were at most *n*.
^	*x*^*y*	Raise *x* to the *y* power. This operation is right-associative, meaning that the right side of the expression is
*	*x**y*	*x* times *y*.
/	*x*/*y*	*x* divided by *y*. This returns an unrounded number.
%	*x*%*y*	*x* modulo *y*. That is, the remainder after *x* is divided by *y*.
+	*x*+*y*	*x* plus *y*.
-	*x*-*y*	*x* minus *y*.
> or gt	*x*>*y*	Check if *x* is greater than *y*. Returns a 1 for yes and 0 for no.
>= or ge	*x*>=*y*	Check if *x* is greater than or equal to *y*. Returns a 1 for yes and 0 for no.
< or lt	*x*<*y*	Check if *x* is less than *y*. Returns a 1 for yes and 0 for no.
<= or le	*x*<=*y*	Check if *x* is less than or equal to *y*. Returns a 1 for yes and 0 for no.
=	*x*=*y*	Check if *x* is equal to *y*. Returns a 1 for yes and 0 for no.

Table

Operator	Expression	Meaning
&	x**&**y	Check if x and y are both nonzero.
|	x|y	Check if at least one of x or y is nonzero.

CHAPTER 3

Library modules

These libraries are mainly for internal use, but are exposed at the top level to be used elsewhere if needed.

3.1 `exceptions`

This is most likely to be used as external applications may want to catch any errors raised by this package.

Exceptions to cover error cases that may be encountered in this package.

A base class for all of them is `RollError`, which means that any function ever thrown by this package can be caught by catching `RollError`.

A `ParseError` is thrown when the initial expression cannot be parsed into an expression tree. Most of these errors occur at the initial tokenization step, but ones that are harder to catch there may be tokenized then fail to construct a valid tree.

An `InputTypeError` indicate that the wrong type of element was passed into one of the main entry point functions like `roll`.

An `EvaluationError` happen when something goes wrong while evaluating the expression tree. These can be split into `ArgumentTypeError` and `ArgumentValueError`, with the same semantics as the builtin `TypeError` and `ValueError`.

exception `dndice.lib.exceptions.`**`ArgumentTypeError`**

Bases: *`dndice.lib.exceptions.EvaluationError`*, `TypeError`

An expression in the roll is of the wrong type.

exception `dndice.lib.exceptions.`**`ArgumentValueError`**

Bases: *`dndice.lib.exceptions.EvaluationError`*, `ValueError`

The value of an expression cannot be used.

exception `dndice.lib.exceptions.`**`EvaluationError`**

Bases: *`dndice.lib.exceptions.RollError`*, `RuntimeError`

The roll could not be evaluated.

exception `dndice.lib.exceptions.`**`InputTypeError`**

Bases: *`dndice.lib.exceptions.RollError`*, `TypeError`

You passed the wrong thing into the entry point function.

exception `dndice.lib.exceptions.`**`ParseError`**(*msg*, *offset*, *expr*)

Bases: *`dndice.lib.exceptions.RollError`*, `ValueError`

The roll expression cannot be parsed into an expression tree.

exception `dndice.lib.exceptions.`**`RollError`**

Bases: `Exception`

A simple base class for all exceptions raised by this module.

3.2 operators

This module holds the operator definitions, and may be worth importing if any extensions are desired.

Defines code representations of arithmetic and rolling operators.

The single most important export from this module is the `OPERATORS` constant. It is a dictionary mapping from the operator codes (think '+', '>=', 'd', etc.) to the actual operator objects.

You might also want to pull out the `Side` enum and `Operator` class if you want to define your own or otherwise do some customization with the operators.

The `Roll` object could be useful if you are trying to extend the rolling functionality.

All the various functions are not really worth talking about, they just implement the operations defined here.

class `dndice.lib.operators.`**`Operator`**(*code: str*, *precedence: int*, *func: Callable*, *arity: dndice.lib.operators.Side = <Side.BOTH: 3>*, *associativity: dndice.lib.operators.Side = <Side.LEFT: 2>*, *cajole: dndice.lib.operators.Side = <Side.BOTH: 3>*, *viewAs: str = None*)

An operator like + or d that can be applied to values.

This class implements a full ordering, but == has very different semantics. The ordering operators (>, <, >=, <=) all compare the precedence of two given operators. == on the other hand compares value/identity, so it is intended to match when comparing two instances of the same operator or, more importantly, comparing an `Operator` to the string that should produce it. For instance, the `Operator` instance for addition should return `True` for `addition == '+'`.

`__call__`(*left*, *right*)

Evaluate the function associated with this operator.

Most operator functions are binary and will consume both `left` and `right`. For unary operators the caller **must** pass in None to fill the unused operand slot. This may be changed in future to be cleverer.

The `cajole` field of this object is used at this stage to collapse one or both operands into a single value. If one of the sides is targeted by the value of `cajole`, and the corresponding operand is a `Roll` or other iterator, it will be replaced by its sum.

Parameters

- **`left`** – The left operand. Usually an int or a Roll.
- **`right`** – The right operand. Even more likely to be an int.

`__init__`(*code: str, precedence: int, func: Callable, arity: dndice.lib.operators.Side = <Side.BOTH: 3>, associativity: dndice.lib.operators.Side = <Side.LEFT: 2>, cajole: dndice.lib.operators.Side = <Side.BOTH: 3>, viewAs: str = None*)

Create a new operator.

Parameters

- **`code`** – The string that represents this operation. For instance, addition is '+' and greater than or equal to is '>='. Since the negative sign and minus operator would be identical in this respect, the sign's `code` differs and is 'm' instead. Similar with positive sign and 'p'.
- **`precedence`** – A higher number means greater precedence. Currently the numbers 1-8 are in use though maybe you can think of more.
- **`func`** – The function performed by this operation. It must take one or two arguments and return one result, with no side effects.
- **`arity`** – Which side(s) this operator draws operands from. For instance, '+' takes arguments on left and right, while '!' takes only one argument on its left.
- **`associativity`** – Which direction the associativity goes. Basically, when precedence is tied, should this be evaluated left to right or right to left. Exponentiation is the only common operation that does the latter.
- **`cajole`** – Which operand(s) should be collapsed into a single value before operation.
- **`viewAs`** – If the code is different than the actual operation string, fill viewAs with the real string.

class `dndice.lib.operators.`**`Roll`**(*rolls=None, die=0*)

A set of rolls.

This tracks the active rolls (those that are actually counted) as well as what die was rolled to get this and any discarded values.

The active rolls are assumed by many of the associated functions to always be sorted ascending. To effect this, a Roll instance will automatically sort the active roll list every time there is an update to it. However, sometimes the index of a particular element does matter, like with the `reroll_unconditional` class of functions that repeatedly perform in-place replacements on those elements. Therefore you have the chance to temporarily disable sorting for the duration of these modifications.

This object can be treated like a list in many ways, implementing get/set/delitem methods, len, and iter.

`__init__`(*rolls=None, die=0*)

Create a new roll.

Parameters

- **`rolls`** – The starting list of values to be used.
- **`die`** – The number of sides of the die that was rolled to get those values.

`copy`() → dndice.lib.operators.Roll

Create a copy of this object to avoid mutating an original.

`discard`(*index: Union[int, slice]*)

Discard a roll or slice of rolls by index.

Parameters **`index`** – The indexing object (int or slice) to select values to discard.

Raises ***`ArgumentTypeError`*** – When something other than int or slice is used.

`replace`(*index, new*)

Discard a roll or slice of rolls and replace with new values.

If you are discarding a slice, you must replace it with a sequence of equal length.

Parameters

- **`index`** – An indexing object, an int or a slice.
- **`new`** – The new value or values to replace the old with.

Raises

- ***`ArgumentTypeError`*** – When something other than int or slice is used for indexing.
- ***`ArgumentValueError`*** – When the size of the replacement doesn't match the size of the slice.

`sorting_disabled``()`

Temporarily disable auto-sorting.

This is a context manager, so is used with the `with` statement. For example:

```
with roll.sorting_disabled():
    while i < len(original):
        while comp(roll[i], target):
            roll.replace(i, single_die(roll.die))
        i += 1
```

The example is taken straight from `reroll_unconditional` below.

`class` `dndice.lib.operators.`**`Side`**

Represents which side an operation is applicable to.

Note that checking if an operation includes one side is as simple as checking `Operator.arity & Side.LEFT` or `Operator.arity & Side.RIGHT`, whichever one you want.

`dndice.lib.operators.`**`ceil_val`**`(`*original: dndice.lib.operators.Roll, top: Union[int, float]*`)` → dndice.lib.operators.Roll

Replace any rolls greater than the given ceiling with that value.

Parameters

- **`original`** – The set of rolls.
- **`top`** – The ceiling to truncate to.

Returns The modified roll set.

`dndice.lib.operators.`**`factorial`**`(`*number: int*`)` → int

Calculate the factorial of a number.

Parameters **`number`** – The argument.

Returns number!

`dndice.lib.operators.`**`floor_val`**`(`*original: dndice.lib.operators.Roll, bottom: Union[int, float]*`)` → dndice.lib.operators.Roll

Replace any rolls less than the given floor with that value.

Parameters

- **`original`** – The set of rolls.
- **`bottom`** – The floor to truncate to.

Returns The modified roll set.

`dndice.lib.operators.`**`reroll_once`**(*original: dndice.lib.operators.Roll, target: Union[int, float], comp: Callable[[Union[int, float], Union[int, float]], bool]*) → dndice.lib.operators.Roll

Take the roll and reroll values that meet the comparison, taking the new result.

Parameters

- **`original`** – The set of rolls to inspect.
- **`target`** – The target to compare against.
- **`comp`** – The comparison function, that should return true if the value should be rerolled.

Returns The roll after performing the rerolls.

`dndice.lib.operators.`**`reroll_once_higher`**(*original: dndice.lib.operators.Roll, target: Union[int, float]*) → dndice.lib.operators.Roll

Reroll and take the new result when a roll is greater than the given number.

`dndice.lib.operators.`**`reroll_once_lower`**(*original: dndice.lib.operators.Roll, target: Union[int, float]*) → dndice.lib.operators.Roll

Reroll and take the new result when a roll is less than the given number.

`dndice.lib.operators.`**`reroll_once_on`**(*original: dndice.lib.operators.Roll, target: Union[int, float]*) → dndice.lib.operators.Roll

Reroll and take the new result when a roll is equal to the given number.

`dndice.lib.operators.`**`reroll_unconditional`**(*original: dndice.lib.operators.Roll, target: Union[int, float], comp: Callable[[Union[int, float], Union[int, float]], bool]*) → dndice.lib.operators.Roll

Reroll values that meet the comparison, and keep on rerolling until they don't.

Parameters

- **`original`** – The set of rolls to inspect.
- **`target`** – The target to compare against.
- **`comp`** – The comparison function, that should return true if the value should be rerolled.

Returns The roll after performing the rerolls.

`dndice.lib.operators.`**`reroll_unconditional_higher`**(*original: dndice.lib.operators.Roll, target: Union[int, float]*) → dndice.lib.operators.Roll

Reroll and keep on rerolling when a roll is greater than the given number.

`dndice.lib.operators.`**`reroll_unconditional_lower`**(*original: dndice.lib.operators.Roll, target: Union[int, float]*) → dndice.lib.operators.Roll

Reroll and keep on rerolling when a roll is less than the given number.

`dndice.lib.operators.`**`reroll_unconditional_on`**(*original: dndice.lib.operators.Roll, target: Union[int, float]*) → dndice.lib.operators.Roll

Reroll and keep on rerolling when a roll is equal to the given number.

`dndice.lib.operators.`**`roll_average`**(*number: int, sides: Union[int, Tuple[float, ...], dndice.lib.operators.Roll]*) → dndice.lib.operators.Roll

Roll an average value on every die.

On most dice this will have a .5 in the result.

`dndice.lib.operators.`**`roll_basic`**(*number: int, sides: Union[int, Tuple[float, ...], dndice.lib.operators.Roll]*) → dndice.lib.operators.Roll

Roll a single set of dice.

Parameters

- **`number`** – The number of dice to be rolled.
- **`sides`** – Roll a `sides`-sided die. Or, if given a collection of side values, pick one from there.

Returns A `Roll` holding all the dice rolls.

`dndice.lib.operators.`**`roll_critical`**(*number: int, sides: Union[int, Tuple[float, ...], dndice.lib.operators.Roll]*) → dndice.lib.operators.Roll

Roll double the normal number of dice.

`dndice.lib.operators.`**`roll_max`**(*number: int, sides: Union[int, Tuple[float, ...], dndice.lib.operators.Roll]*) → dndice.lib.operators.Roll

Roll a maximum value on every die.

`dndice.lib.operators.`**`single_die`**(*sides: Union[int, Tuple[float, ...], dndice.lib.operators.Roll]*) → Union[int, float]

Roll a single die.

The behavior is different based on what gets passed in. Given an int, it rolls a die with that many sides (precisely, it returns a random number between 1 and `sides` inclusive). Given a tuple, meaning the user specified a particular set of values for the sides of the die, it returns one of those values selected at random. Given a `Roll`, which can happen in weird cases like 2d(1d4), it will take the sum of that roll and use it as the number of sides of a die.

Parameters **`sides`** – The number of sides, or specific side values.

Returns The random value that was rolled.

`dndice.lib.operators.`**`take_high`**(*roll: dndice.lib.operators.Roll, number: int*) → dndice.lib.operators.Roll

Preserve the highest [number] rolls and discard the rest.

This is used to implement advantage in D&D 5e.

Parameters

- **`roll`** – The set of rolls.
- **`number`** – The number of rolls to take.

Returns A roll with the highest rolls preserved and the rest discarded.

`dndice.lib.operators.`**`take_low`**(*roll: dndice.lib.operators.Roll, number: int*) → dndice.lib.operators.Roll

Preserve the lowest [number] rolls and discard the rest.

This is used to implement disadvantage in D&D 5e.

Parameters

- **`roll`** – The set of rolls.
- **`number`** – The number of rolls to take.

Returns A roll with the lowest rolls preserved and the rest discarded.

`dndice.lib.operators.`**`threshold_lower`**(*roll: dndice.lib.operators.Roll, threshold: int*) → dndice.lib.operators.Roll

Count the rolls that are equal to or above the given threshold.

Parameters

- **`roll`** – The set of rolls.
- **`threshold`** – The number to compare against.

Returns A list of ones and zeros that indicate which rolls met the threshold.

`dndice.lib.operators.`**`threshold_upper`**(*roll: dndice.lib.operators.Roll*, *threshold: int*) → dndice.lib.operators.Roll

Count the rolls that are equal to or below the given threshold.

Parameters

- **`roll`** – The set of rolls.
- **`threshold`** – The number to compare against.

Returns A list of ones and zeros that indicate which rolls met the threshold.

`dndice.lib.operators.`**`OPERATORS = {'!': !:l 8, '%': %:lr 3, '&': &:lr 1, '*': *:lr 3, '`

This contains all of the operators that are actually defined by this module. It is a map between the codes used to represent the operators, and the actual `Operator` instances that hold the functionality. Visually, it is sorted in descending order of precedence. Obviously, as dictionaries are unsorted, this doesn't actually matter.

3.3 `evaltree`

This module implements an expression tree for use by the algorithms in this package, and may be imported to extend build on existing functionality.

Classes to hold and work with evaluation trees.

`EvalTree` is naturally the core class here. It provides all the functionality for looking at the tree as a unit, while `EvalTreeNode` is the basic component.

`class` `dndice.lib.evaltree.`**`EvalTree`**(*source: Union[str, List[Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str]], EvalTree, None]*)

An expression tree that can be used to evaluate roll expressions.

An expression tree is, in short, a binary tree that holds an arithmetic expression. It is important to note that the binary tree not necessarily be complete; unary operators like factorial do create a tree where some leaf slots are unfilled, as they only take one operand instead of the two that most operators take.

It is guaranteed that all leaf nodes hold a value (usually an integer) while all non-leaf nodes hold an operator.

`__add__`(*other*) → dndice.lib.evaltree.EvalTree

Join two trees together with the addition operator.

The end result is as if you had wrapped both initial expressions in parentheses and added them together, like `(expression 1) + (expression 2)`.

Parameters **`other`** – The EvalTree to join with this one.

Raises **`NotImplementedError`** – If anything but an EvalTree is passed in.

`__iadd__`(*other*) → dndice.lib.evaltree.EvalTree

Join two trees together with the addition operator in-place.

The end result is as if you had wrapped both initial expressions in parentheses and added them together, like `(expression 1) + (expression 2)`.

As this mutates the objects in-place instead of performing a deep clone, it is much faster than the normal addition. This comes with a **very important warning**, though: neither of the inputs ends up independent of the output. If you use this operator, don't use either argument to it again. Otherwise the evaluation of any of the objects will pollute the others.

Parameters **`other`** – The EvalTree to join with this one.

Raises **`NotImplementedError`** – If anything but an EvalTree is passed in.

`__init__`(*source: Union[str, List[Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str]], EvalTree, None]*)

Initialize a tree of EvalTreeNodes that represent a given expression.

Parameters **`source`** – The expression, generally as a string or already tokenized list or compiled tree.

`__isub__`(*other*)

Join two trees together with the subtraction operator in-place.

The end result is as if you had wrapped both initial expressions in parentheses and subtracted them, like `(expression 1) - (expression 2)`.

As this mutates the objects in-place instead of performing a deep clone, it is much faster than the normal subtraction. This comes with a **very important warning**, though: neither of the inputs ends up independent of the output. If you use this operator, don't use either argument to it again. Otherwise the evaluation of any of the objects will pollute the others.

Parameters **`other`** – The EvalTree to join with this one.

Raises **`NotImplementedError`** – If anything but an EvalTree is passed in.

`__repr__`()

Produce a string that can be used to reconstruct this tree.

`__sub__`(*other*)

Join two trees together with the subtraction operator.

The end result is as if you had wrapped both initial expressions in parentheses and subtracted them, like `(expression 1) - (expression 2)`.

Parameters **`other`** – The EvalTree to join with this one.

Raises **`NotImplementedError`** – If anything but an EvalTree is passed in.

`averageify`() → dndice.lib.evaltree.EvalTree

Modify rolls in this expression to average rolls.

Returns This tree after it has been modified in-place.

`critify`() → dndice.lib.evaltree.EvalTree

Modify rolls in this expression to critical rolls.

Returns This tree after it has been modified in-place.

`evaluate`() → Union[int, float]

Recursively evaluate the tree.

Along the way, the `value` of each node is set to the value of the expression at this stage, so it can be inspected later. This is used to great effect by the "verbose mode" of the main roll function.

What is meant by "value of the expression at this stage" can be shown through a diagram:

```
      -         < 0
    /   \
  *       +     < 1
/   \   /   \
4   5   1   2   < 2
```

This is the tree that would result from the expression "4 * 5 - (1 + 2)". If we were to start evaluating this tree, we would first recursively run down all three levels. Once reaching the leaves at level 2, their value is obvious: they are concrete already. Copy their `payload` into their `value`. One level up, and we reach operators. The operator nodes receive values from each of their children, perform the operation they hold, and fill their own `value` slot with the result. For instance, the '*' would perform 4 * 5 and store 20. This continues until the root is reached, and the final value is returned.

Returns The single final value from the tree.

`in_order`(*abort=None*) → Iterable[dndice.lib.evaltree.EvalTreeNode]
Perform an in-order/infix traversal of the tree.

This includes a minimal set of parentheses such that the original tree can be constructed from the results of this iterator.

`is_critical`() → bool
Checks if this roll contains a d20 roll that is a natural 20.

`is_fail`() → bool
Checks if this roll contains a d20 roll that is a natural 1.

`maxify`() → dndice.lib.evaltree.EvalTree
Modify rolls in this expression to maximum rolls.

Returns This tree after it has been modified in-place.

`pre_order`(*abort=None*) → Iterable[dndice.lib.evaltree.EvalTreeNode]
Perform a pre-order/breadth-first traversal of the tree.

`verbose_result`() → str
Forms an infix string of the result, looking like the original with rolls evaluated.

The expression is constructed with as few parentheses as possible. This means that if there were redundant parentheses in the input, they will not show up here.

Returns A string representation of the result, showing the results from rolls.

class `dndice.lib.evaltree.`**`EvalTreeNode`**(*payload: Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str, None], left: Optional[dndice.lib.evaltree.EvalTreeNode] = None, right: Optional[dndice.lib.evaltree.EvalTreeNode] = None*)
A node in the EvalTree, which can hold a value or operator.

`__init__`(*payload: Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str, None], left: Optional[dndice.lib.evaltree.EvalTreeNode] = None, right: Optional[dndice.lib.evaltree.EvalTreeNode] = None*)
Initialize a new node in the expression tree.

Leaf nodes (those with no left or right children) are guaranteed to hold concrete values while non-leaf nodes are guaranteed to hold operators.

Parameters

- **`payload`** – The operator or value that is expressed by this node.

- **left** – The left child of this node, which holds the operand or expression to the left of this operator.
- **right** – The right child of this node, which holds the operand or expression to the right of this operator.

__repr__()
Produce a string that could be used to reconstruct this node.

evaluate() → Union[dndice.lib.operators.Roll, int, float]
Recursively evaluate this subtree and return its computed value.

As a side effect, it also annotates this node with the value. At the EvalTree level, this can be used to compose a more detailed report of the dice rolls.

Returns The value computed.

3.4 helpers

This module is for a few helpful decorators, which have probably been implemented elsewhere already.

A few helper decorators to simplify construction of the code.

`dndice.lib.helpers.`**check_simple_types**(*f: Callable*) → Callable
A decorator that will check the types of the arguments at runtime.

This has a limitation that it can only deal with a concrete number of positional arguments, each of which is annotated with a concrete type. This means that no fancy *typing* annotations will be supported.

`dndice.lib.helpers.`**wrap_exceptions_with**(*ex: type*, *message=”*, *target=<class ’Exception’>*)
Catch exceptions and rethrow them wrapped in an exception of our choosing.

This is mainly for the purpose of catching whatever builtin exceptions might be thrown and showing them to the outside world as custom module exceptions. That way an external importer can just catch the RollError and thus catch every exception thrown by this module.

`target` specifies what to catch and therefore wrap. By default it’s `Exception` to cast a wide net but note that you can catch any specific exception you please, or a set of exceptions if you pass in a tuple of exception classes.

And yes, this is just try/catch/rethrow. It’s more decorative this way though. Plus saves you some indentation as you’re trying to wrap an entire function.

Parameters

- **ex** – The exception to use as wrapper.
- **message** – A custom message to use for the wrapper exception.
- **target** – Which exception(s) you want to catch.

3.5 tokenizer

Split a string into a list of tokens, meaning operators or values.

Two functions may be useful to the outside: `tokens` and `tokens_lazy`. `tokens` returns a list of the tokens, while `tokens_lazy` is a generator `dndice.lib.tokenizer.`**`tokens`**(*s: str*) → List[Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str]]

Splits an expression into tokens that can be parsed into an expression tree.

For specifics, see `tokens_lazy()`.

Parameters **`s`** – The expression to be parsed

Returns A list of tokens

`dndice.lib.tokenizer.`**`tokens_lazy`**(*s: str*) → Iterable[Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str]]

Splits an expression into tokens that can be parsed into an expression tree.

This parser is based around a state machine. Starting with InputStart, it traverses the string character by character taking on a sequence of states. At each of these states, it asks that state to produce a token if applicable and produce the state that follows it. The token is yielded and the movement of the machine continues.

Parameters **`s`** – The expression to be parsed

Returns An iterator of tokens

`class` `dndice.lib.tokenizer.`**`State`**(*expr: str*, *i: int*)

The base class for all of the states of the tokenizer.

A number of these states will consume one or more characters to produce a token, but some instead serve as markers. For instance, the ExprStart state marks the start of an expression, which can be followed by a unary prefix operator, an open parenthesis (which will itself start a new expression), or an integer value.

This base class also provides several constants and simple methods that factor into the algorithms of most or all varieties of states.

Variables

- **`_recognized`** – The set of all characters that could be part of an expression.
- **`starters`** – The set of all characters that can start this token.
- **`consumes`** – The maximum number of characters this token can consume.
- **`followers`** – The set of states that are allowed to follow this one. This is a sequence of the class objects. It must be ordered, and if applicable, 'ExprStart'/ 'ExprEnd' must come last. Otherwise, anything could cause a transfer to those marker states. This should semantically be a class variable; it is not simply because then it would be evaluated at definition time and it needs to reference other class objects which don't exist at that time.

`run`() → Tuple[Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str, None], dndice.lib.tokenizer.State]

Moves along the string to produce the current token.

This base implementation moves along the string, performing the following steps:

1. If the current character is not one that is recognized, raise a ParseError.
2. If the current token is whitespace, run along the string until the end of the whitespace, then transfer to the next state. If the `next_state` function indicates that the character after the whitespace should be part of the existing one, raise a ParseError.
3. Check if the machine should transfer to the next state using the `next_state` function. If yes, return the token that has been collected along with that next state.

It stops when it encounters the first character of the next token.

Raises ***`ParseError`*** – If a character is not recognized, or a token is not allowed in a certain position, or the expression ends unexpectedly.

Returns The pair (the token that this produces if applicable, the next state)

next_state(*char: str, agg: Sequence[str]*) → Optional[dndice.lib.tokenizer.State]
Decides what state to transfer to.

Parameters

- **char** – The current character.
- **agg** – The sequence of characters that has been identified as the current token.

Returns An instance of the State that will pick up starting with the current character.

collect(*agg: Sequence[str]*) → Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str, None]
Create a token from the current aggregator.

Returns None if this state does not produce a token. This is for the "marker" states that correspond to abstract notions of positions in the expression rather than actual concrete tokens.

This default implementation simply returns None, which is common to all of the marker states while every real token will have its own implementation.

Parameters **agg** – The list of characters that compose this token.

class dndice.lib.tokenizer.**Integer**(*expr: str, i: int*)
A whole number.

Is followed by the end of an expression.

collect(*agg: List[str]*) → Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str]
Collects the sequence of characters into an int.

class dndice.lib.tokenizer.**Operator**(*expr: str, i: int*)
An incomplete base class for the operators.

collect(*agg: Sequence[str]*) → Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str, None]
Interprets the sequence of characters as an operator.

Raises ***ParseError*** – If the characters don't actually compose a real operator.

class dndice.lib.tokenizer.**Binary**(*expr: str, i: int*)
A binary operator.

Is followed by the start of an expression.

next_state(*char: str, agg: Sequence[str] = None*) → Optional[dndice.lib.tokenizer.State]
Reads a possibly multi-character operator.

Overrides because unlike integers, they have finite length and the order is important, and unlike other tokens like parentheses, they may have length greater than one. This requires checking at every step whether the appending of the current character to the current sequence produces a valid operator.

class dndice.lib.tokenizer.**UnaryPrefix**(*expr: str, i: int*)
A unary prefix operator.

The only ones that exist now are the positive and negative signs.

Is followed by the start of an expression.

collect(*agg: Sequence[str]*) → Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str, None]
Returns the correct unary operator.

These are special cases to avoid ambiguity in the definitions.

class `dndice.lib.tokenizer.`**`UnarySuffix`**(*expr: str*, *i: int*)

A unary suffix operator.

The only one that exists now is the factorial, !.

Is followed by the end of an expression.

class `dndice.lib.tokenizer.`**`Die`**(*expr: str*, *i: int*)

One of the die operators, which are a subset of the binary.

Can be followed by:

- A number
- A list
- The "fudge die"
- An open parenthesis. Note that this will not remember that this is the sides of a die and will not therefore allow the special tokens that can only exist as the sides of dice: the fudge die and the side list.

class `dndice.lib.tokenizer.`**`FudgeDie`**(*expr: str*, *i: int*)

The "fudge die" value.

Followed by the end of the expression or string.

`collect`(*agg: Sequence[str]*) → Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str, None]

Produces the side list [-1, 0, 1].

`next_state`(*char: str*, *agg: Sequence[str] = None*) → Optional[dndice.lib.tokenizer.State]

Goes directly to the end of the expression.

class `dndice.lib.tokenizer.`**`ListToken`**(*expr: str*, *i: int*)

Collects a list of die sides into a single token.

`run`() → Tuple[Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str], dndice.lib.tokenizer.State]

Uses a sub-state machine to read the list.

class `dndice.lib.tokenizer.`**`ListStart`**(*expr: str*, *i: int*)

Starts the list.

Can be followed by a value.

class `dndice.lib.tokenizer.`**`ListValue`**(*expr: str*, *i: int*)

A value in a sides list.

Can be followed by a list separator or the end of the list.

`collect`(*agg: Sequence[str]*) → Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str, float, None]

Collects the sequence of characters into a float.

class `dndice.lib.tokenizer.`**`ListSeparator`**(*expr: str*, *i: int*)

The comma that separates the list values.

Can only be followed by a value.

class `dndice.lib.tokenizer.`**`ListEnd`**(*expr: str*, *i: int*)

Ends the list.

Followed by the end of the expression or string.

class `dndice.lib.tokenizer.`**`OpenParen`**(*expr: str*, *i: int*)

An open parenthesis.

Is followed by the start of an expression.

next_state(*char: str*, *agg: Sequence[str] = None*) → Optional[dndice.lib.tokenizer.State]
Goes directly into the start of an expression.

collect(*agg: List[str]*) → Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str, None]
Produces the single character '('.

class dndice.lib.tokenizer.**CloseParen**(*expr: str*, *i: int*)
An closing parenthesis.

Is followed by the end of an expression.

next_state(*char: str*, *agg: Sequence[str] = None*) → Optional[dndice.lib.tokenizer.State]
Goes directly into the end of an expression.

collect(*agg: List[str]*) → Union[dndice.lib.operators.Roll, int, Tuple[float, ...], dndice.lib.operators.Operator, str, None]
Produces the single character '('.

class dndice.lib.tokenizer.**ExprStart**(*expr: str*, *i: int*)
The start of a subexpression.

Can be followed by:

- An open parenthesis (which also opens a new expression)
- A unary prefix operator (the negative and positive marks)
- A number

class dndice.lib.tokenizer.**ExprEnd**(*expr: str*, *i: int*)
The end of a subexpression.

Can be followed by:

- A unary suffix operator (the only existing one is !)
- A binary operator
- A die expression (which is a subset of the binary operators)
- A close parenthesis (which also terminates an enclosing subexpression)
- The end of the input string

class dndice.lib.tokenizer.**InputStart**(*expr: str*, *i: int*)
The start of the string.

Can be followed by:

- The end of the input string, leading to an empty sequence of tokens.
- The start of an expression.

class dndice.lib.tokenizer.**InputEnd**(*expr: str*, *i: int*)
The end of the string.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index