d20

Andrew Zhu, D&D Beyond

Jul 08, 2021

CONTENTS:

CHAPTER

ONE

D20

A fast, powerful, and extensible dice engine for D&D, d20 systems, and any other system that needs dice!

1.1 Key Features

- Quick to start - just use `d20.roll()`!
- Optimized for speed and memory efficiency
- Highly extensible API for custom behaviour and dice stringification
- Built-in execution limits against malicious dice expressions
- Tree-based dice representation for easy traversal

1.2 Installing

Requires Python 3.6+.

```
python3 -m pip install -U d20
```

1.3 Quickstart

```
>>> import d20
>>> result = d20.roll("1d20+5")
>>> str(result)
'1d20 (10) + 5 = `15`'
>>> result.total
15
```

(continues on next page)

(continued from previous page)

```
>>> result.crit
<CritType.NORMAL: 0>
>>> str(result.ast)
'1d20 + 5'
```

1.4 Documentation

Check out the docs on Read the Docs!

1.5 Dice Syntax

This is the grammar supported by the dice parser, roughly ordered in how tightly the grammar binds.

1.5.1 Numbers

These are the atoms used at the base of the syntax tree.

Name	Syntax	Description	Examples
literal	`INT`, `DECIMAL`	A literal number.	`1`, `0.5`, `3.14`
dice	`INT? "d" (INT | "%")`	A set of die.	`d20`, `3d6`
set	`"(" (num "," num)* ","?)? ")"`	A set of expressions.	`()`, `(2,)`, `(1, 3+3, 1d20)`

Note that `(3d6)` is equivalent to `3d6`, but `(3d6,)` is the set containing the one element `3d6`.

1.5.2 Set Operations

These operations can be performed on dice and sets.

Grammar

Name	Syntax	Description	Examples
set_op	`operation selector`	An operation on a set (see below).	`kh3`, `ro<3`
selector	`seltype INT`	A selection on a set (see below).	`3`, `h1`, `>2`

Operators

Operators are always followed by a selector, and operate on the items in the set that match the selector.

Syntax	Name	Description
k	keep	Keeps all matched values.
p	drop	Drops all matched values.
rr	reroll	Rerolls all matched values until none match. (Dice only)
ro	reroll once	Rerolls all matched values once. (Dice only)
ra	reroll and add	Rerolls up to one matched value once, keeping the original roll. (Dice only)
e	explode on	Rolls another die for each matched value. (Dice only)
mi	minimum	Sets the minimum value of each die. (Dice only)
ma	maximum	Sets the maximum value of each die. (Dice only)

Selectors

Selectors select from the remaining kept values in a set.

Syntax	Name	Description
X	literal	All values in this set that are literally this value.
hX	highest X	The highest X values in the set.
lX	lowest X	The lowest X values in the set.
>X	greater than X	All values in this set greater than X.
<X	less than X	All values in this set less than X.

1.5.3 Unary Operations

Syntax	Name	Description
+X	positive	Does nothing.
-X	negative	The negative value of X.

1.5.4 Binary Operations

Syntax	Name
X * Y	multiplication
X / Y	division
X // Y	int division
X % Y	modulo
X + Y	addition
X - Y	subtraction
X == Y	equality
X >= Y	greater/equal
X <= Y	less/equal
X > Y	greater than
X < Y	less than
X != Y	inequality

1.5.5 Examples

```
>>> from d20 import roll
>>> r = roll("4d6kh3")  # highest 3 of 4 6-sided dice
>>> r.total
14
>>> str(r)
'4d6kh3 (4, 4, **6**, ~~3~~) = `14`'

>>> r = roll("2d6ro<3")  # roll 2d6s, then reroll any 1s or 2s once
>>> r.total
9
>>> str(r)
'2d6ro<3 (**~~1~~**, 3, **6**) = `9`'

>>> r = roll("8d6mi2")  # roll 8d6s, with each die having a minimum roll of 2
>>> r.total
33
>>> str(r)
'8d6mi2 (1 -> 2, **6**, 4, 2, **6**, 2, 5, **6**) = `33`'

>>> r = roll("(1d4 + 1, 3, 2d6kl1)kh1")  # the highest of 1d4+1, 3, and the lower of␣
↪2 d6s
>>> r.total
3
>>> str(r)
'(1d4 (2) + 1, ~~3~~, ~~2d6kl1 (2, 5)~~)kh1 = `3`'
```

1.6 Custom Stringifier

By default, d20 stringifies the result of each dice roll formatted in Markdown, which may not be useful in your application. To change this behaviour, you can create a subclass of ``*d20.Stringifier*` <https://github.com/avrae/d20/blob/master/d20/stringifiers.py>`_ (or `d20.SimpleStringifier` as a starting point), and implement the `_str_*` methods to customize how your dice tree is stringified.

Then, simply pass an instance of your stringifier into the `roll()` function!

```
>>> import d20
>>> class MyStringifier(d20.SimpleStringifier):
...     def _stringify(self, node):
...         if not node.kept:
...             return 'X'
...         return super()._stringify(node)
...
...     def _str_expression(self, node):
...         return f"The result of the roll {self._stringify(node.roll)} was
↪{int(node.total)}"

>>> result = d20.roll("4d6e6kh3", stringifier=MyStringifier())
>>> str(result)
'The result of the roll 4d6e6kh3 (X, 5, 6!, 6!, X, X) was 17'
```

1.7 Annotations and Comments

Each dice node supports value annotations - i.e., a method to "tag" parts of a roll with some indicator. For example,

```
>>> from d20 import roll
>>> str(roll("3d6 [fire] + 1d4 [piercing]"))
'3d6 (3, 2, 2) [fire] + 1d4 (3) [piercing] = `10`'

>>> str(roll("-(1d8 + 3) [healing]"))
'-(1d8 (7) + 3) [healing] = `-10`'

>>> str(roll("(1 [one], 2 [two], 3 [three])"))
'(1 [one], 2 [two], 3 [three]) = `6`'
```

are all examples of valid annotations. Annotations are purely visual and do not affect the evaluation of the roll by default.

Additionally, when `allow_comments=True` is passed to `roll()`, the result of the roll may have a comment:

```
>>> from d20 import roll
>>> result = roll("1d20 I rolled a d20", allow_comments=True)
>>> str(result)
'1d20 (13) = `13`'
>>> result.comment
'I rolled a d20'
```

Note that while `allow_comments` is enabled, AST caching is disabled, which may lead to slightly worse performance.

1.8 Traversing Dice Results

The raw results of dice rolls are returned in ``*Expression*` <https://github.com/avrae/d20/blob/master/d20/models.py#L76>`_ objects, which can be accessed as such:

```
>>> from d20 import roll
>>> result = roll("3d6 + 1d4 + 3")
>>> str(result)
'3d6 (4, **6**, **6**) + 1d4 (**1**) + 3 = `20`'
>>> result.expr
<Expression roll=<BinOp left=<BinOp left=<Dice num=3 size=6 values=[<Die size=6␣
↪values=[<Literal 4>]>, <Die size=6 values=[<Literal 6>]>, <Die size=6 values=[
↪<Literal 6>]>] operations=[]> op=+ right=<Dice num=1 size=4 values=[<Die size=4␣
↪values=[<Literal 1>]>] operations=[]>> op=+ right=<Literal 3>> comment=None>
```

or, in a easier-to-read format,

```
<Expression
    roll=<BinOp
        left=<BinOp
            left=<Dice
                num=3
                size=6
                values=[
                    <Die size=6 values=[<Literal 4>]>,
                    <Die size=6 values=[<Literal 6>]>,
```

(continues on next page)

(continued from previous page)

```
                    <Die size=6 values=[<Literal 6>]>
                ]
                operations=[]
            >
            op=+
            right=<Dice
                num=1
                size=4
                values=[
                    <Die size=4 values=[<Literal 1>]>
                ]
                operations=[]
            >
        >
        op=+
        right=<Literal 3>
    >
    comment=None
>
```

From here, `Expression.children` returns a tree of nodes representing the expression from left to right, each of which may have children of their own. This can be used to easily search for specific dice, look for the left-most operand, or modify the result by adding in resistances or other modifications.

1.8.1 Examples

Finding the left and right-most operands:

```
>>> from d20 import roll

>>> binop = roll("1 + 2 + 3 + 4")
>>> left = binop.expr
>>> while left.children:
...     left = left.children[0]
>>> left
<Literal 1>

>>> right = binop.expr
>>> while right.children:
...     right = right.children[-1]
>>> right
<Literal 4>

>>> from d20 import utils  # these patterns are available in the utils submodule:
>>> utils.leftmost(binop.expr)
<Literal 1>
>>> utils.rightmost(binop.expr)
<Literal 4>
```

Searching for the d4:

```
>>> from d20 import roll, Dice, SimpleStringifier, utils

>>> mixed = roll("-1d8 + 4 - (3, 1d4)kh1")
>>> str(mixed)
```

(continues on next page)

(continued from previous page)

```
'-1d8 (**8**) + 4 - (3, ~~1d4 (3)~~)kh1 = `-7`'
>>> root = mixed.expr
>>> result = utils.dfs(root, lambda node: isinstance(node, Dice) and node.num == 1␣
↪and node.size == 4)
>>> result
<Dice num=1 size=4 values=[<Die size=4 values=[<Literal 3>]>] operations=[]>
>>> SimpleStringifier().stringify(result)
'1d4 (3)'
```

As a note, even though a `Dice` object is the parent of `Die` objects, `Dice.children` returns an empty list, since it's more common to look for the dice, and not each individual component of that dice.

1.9 Performance

By default, the parser caches the 256 most frequently used dice expressions in an LFU cache, allowing for a significant speedup when rolling many of the same kinds of rolls. This caching is disabled when `allow_comments` is True.

With caching:

```
$ python3 -m timeit -s "from d20 import roll" "roll('1d20')"
10000 loops, best of 5: 21.6 usec per loop
$ python3 -m timeit -s "from d20 import roll" "roll('100d20')"
500 loops, best of 5: 572 usec per loop
$ python3 -m timeit -s "from d20 import roll; expr='1d20+'*50+'1d20'" "roll(expr)"
500 loops, best of 5: 732 usec per loop
$ python3 -m timeit -s "from d20 import roll" "roll('10d20rr<20')"
1000 loops, best of 5: 1.13 msec per loop
```

Without caching:

```
$ python3 -m timeit -s "from d20 import roll" "roll('1d20')"
5000 loops, best of 5: 61.6 usec per loop
$ python3 -m timeit -s "from d20 import roll" "roll('100d20')"
500 loops, best of 5: 620 usec per loop
$ python3 -m timeit -s "from d20 import roll; expr='1d20+'*50+'1d20'" "roll(expr)"
500 loops, best of 5: 2.1 msec per loop
$ python3 -m timeit -s "from d20 import roll" "roll('10d20rr<20')"
1000 loops, best of 5: 1.26 msec per loop
```

CHAPTER

TWO

CLASS REFERENCE

Looking for information on *d20.Expression* or *d20.ast.Node*? Check out *Expression Tree* for information on the Expression returned by a roll, or *Abstract Syntax Tree* for the AST returned by a parse.

2.1 Dice

class d20.**Roller**(*context: Optional[d20.dice.RollContext] = None*)

The main class responsible for parsing dice into an AST and evaluating that AST.

context: *d20.RollContext*

The class used to track roll limits.

parse(*expr: str*, *allow_comments: bool = False*) → d20.diceast.Expression

Parses a dice expression into an AST.

Parameters

- **expr** (*str*) – The dice to roll.
- **allow_comments** (*bool*) – Whether to parse for comments after the main roll expression (potential slowdown)

Return type *ast.Expression*

roll(*expr: Union[str, ASTNode], stringifier: Optional[d20.stringifiers.Stringifier] = None, allow_comments: bool = False, advantage: d20.dice.AdvType = <AdvType.NONE: 0>*) → d20.dice.RollResult

Rolls the dice.

Parameters

- **expr** (*str or ast.Node*) – The dice to roll.
- **stringifier** (*d20.Stringifier*) – The stringifier to stringify the result. Defaults to MarkdownStringifier.
- **allow_comments** (*bool*) – Whether to parse for comments after the main roll expression (potential slowdown)
- **advantage** (*AdvType*) – If the roll should be made at advantage. Only applies if the leftmost node is 1d20.

Return type *RollResult*

class d20.**RollResult**(*the_ast: ASTNode*, *the_roll: d20.expression.Expression*, *stringifier: d20.stringifiers.Stringifier*)

Holds information about the result of a roll. This should generally not be constructed manually.

ast: *d20.ast.Node*
The abstract syntax tree of the dice expression that was rolled.

expr: *d20.Expression*
The Expression representation of the result of the roll.

total: int
The integer result of the roll (rounded towards 0).

result: str
The string result of the roll. Equivalent to `stringifier.stringify(self.expr)`.

comment: str or None
If `allow_comments` was True and a comment was found, the comment. Otherwise, None.

property crit
If the leftmost node was Xd20kh1, returns *CritType.CRIT* if the roll was a 20 and *CritType.FAIL* if the roll was a 1. Returns *CritType.NONE* otherwise.

Return type *CritType*

class d20.**RollContext**(*max_rolls=1000*)
A class to track information about rolls to ensure all rolls halt eventually.

To use this class, pass an instance to the constructor of *d20.Roller*.

count_roll(*n=1*)
Called each time a die is about to be rolled.

Parameters **n** (*int*) – The number of rolls about to be made.

Raises *d20.TooManyRolls* – if the roller should stop rolling dice because too many have been rolled.

reset()
Called at the start of each new roll.

class d20.**AdvType**(*value*)
Integer enumeration representing at what advantage a roll should be made at.

NONE: int
Equivalent to `0`. Represents no advantage.

ADV: int
Equivalent to `1`. Represents advantage.

DIS: int
Equivalent to `-1`. Represents disadvantage.

class d20.**CritType**(*value*)
Integer enumeration representing the crit type of a roll.

NONE: int
Equivalent to `0`. Represents no crit.

CRIT: int
Equivalent to `1`. Represents a critical hit (a natural 20 on a d20).

FAIL: int
Equivalent to `2`. Represents a critical fail (a natural 1 on a d20).

2.2 Stringifiers

class d20.**Stringifier**

ABC for string builder from dice result. Children should implement all `_str_*` methods to transform an Expression into a str.

stringify(*the_roll: ExpressionNode*) → str

Transforms a rolled expression into a string recursively, bottom-up.

Parameters **the_roll** (d20.Expression) – The expression to stringify.

Return type str

_stringify(*node: ExpressionNode*) → str

Called on each node that needs to be stringified.

Parameters **node** (d20.Number) – The node to stringify.

Return type str

_str_expression(*node: d20.expression.Expression*) → str

Parameters **node** (d20.Expression) – The node to stringify.

Return type str

_str_literal(*node: d20.expression.Literal*) → str

Parameters **node** (d20.Literal) – The node to stringify.

Return type str

_str_unop(*node: d20.expression.UnOp*) → str

Parameters **node** (d20.UnOp) – The node to stringify.

Return type str

_str_binop(*node: d20.expression.BinOp*) → str

Parameters **node** (d20.BinOp) – The node to stringify.

Return type str

_str_parenthetical(*node: d20.expression.Parenthetical*) → str

Parameters **node** (d20.Parenthetical) – The node to stringify.

Return type str

_str_set(*node: d20.expression.Set*) → str

Parameters **node** (d20.Set) – The node to stringify.

Return type str

_str_dice(*node: d20.expression.Dice*) → str

Parameters **node** (d20.Dice) – The node to stringify.

Return type str

_str_die(*node: d20.expression.Die*) → str

Parameters **node** (d20.Die) – The node to stringify.

Return type str

class d20.**SimpleStringifier**
Example stringifier.

class d20.**MarkdownStringifier**
Transforms roll expressions into Markdown.

2.3 Errors

class d20.**RollError**(*msg*)
Generic exception happened in the roll. Base exception for all library exceptions.

class d20.**RollSyntaxError**(*line*, *col*, *got*, *expected*)
Syntax error happened while parsing roll.

class d20.**RollValueError**(*msg*)
A bad value was passed to an operator.

class d20.**TooManyRolls**(*msg*)
Too many dice rolled (in an individual dice or in rerolls).

CHAPTER

THREE

EXPRESSION TREE

This page documents the structure of the Expression object as returned by `roll(...).expr`. If you're looking for the Expression object returned by `parse(...)`, check out *Abstract Syntax Tree*.

class d20.**Number**(*kept=True*, *annotation=None*)

Bases: abc.ABC, d20.diceast.ChildMixin

The base class for all expression objects.

Note that Numbers implement all the methods of a *ChildMixin*.

kept: bool

Whether this Number was kept or dropped in the final calculation.

annotation: Optional[str]

The annotation on this Number, if any.

property children

(read-only) The children of this Number, usually used for traversing the expression tree.

Return type list[*Number*]

property left

The leftmost child of this Number, usually used for traversing the expression tree.

Return type *Number*

property right

The rightmost child of this Number, usually used for traversing the expression tree.

Return type *Number*

set_child(*index*, *value*)

Sets the ith child of this Number.

Parameters

- **index** (*int*) – Which child to set.
- **value** (Number) – The Number to set it to.

drop()

Makes the value of this Number node not count towards a total.

property keptset

Returns the set representation of this object, but only including children whose values were not dropped.

Return type list[*Number*]

property number

Returns the numerical value of this object.

Return type int or float

property set
Returns the set representation of this object.

Return type list[*Number*]

property total
Returns the numerical value of this object with respect to whether it's kept. Generally, this is preferred to use over `number`, as this will return 0 if the number node was dropped.

Return type int or float

class d20.**Expression**(*roll*, *comment*, ***kwargs*)
Bases: d20.expression.Number

Expressions are usually the root of all Number trees.

roll: *d20.Number*
The roll of this expression.

comment: **Optional[str]**
The comment of this expression.

class d20.**Literal**(*value*, ***kwargs*)
Bases: d20.expression.Number

A literal integer or float.

values: **list[Union[int, float]]**
The history of numbers that this literal has been.

exploded: **bool**
Whether this literal was a value in a *Die* object that caused it to explode.

explode()
Marks that this literal was a value in a *Die* object that caused it to explode.

update(*value*)
Changes the literal value this literal represents.

Parameters **value** (*Union[int, float]*) – The new value.

class d20.**UnOp**(*op*, *value*, ***kwargs*)
Bases: d20.expression.Number

Represents a unary operation.

op: **str**
The unary operation.

value: *d20.Number*
The subtree that the operation operates on.

class d20.**BinOp**(*left*, *op*, *right*, ***kwargs*)
Bases: d20.expression.Number

Represents a binary operation.

op: **str**
The binary operation.

left: *d20.Number*
The left subtree that the operation operates on.

right: *d20.Number*
The right subtree that the operation operates on.

class d20.**Parenthetical**(*value*, *operations=None*, ***kwargs*)
Bases: d20.expression.Number

Represents a value inside parentheses.

value: *d20.Number*
The subtree inside the parentheses.

operations: list[d20.SetOperation]
If the value inside the parentheses is a *Set*, the operations to run on it.

class d20.**Set**(*values*, *operations=None*, ***kwargs*)
Bases: d20.expression.Number

Represents a set of values.

values: list[*d20.Number*]
The elements of the set.

operations: list[d20.SetOperation]
The operations to run on the set.

class d20.**Dice**(*num*, *size*, *values*, *operations=None*, *context=None*, ***kwargs*)
Bases: d20.expression.Set

A set of Die.

num: int
How many *Die* in this set of dice.

size: int
The size of each *Die* in this set of dice.

values: list[*d20.Die*]
The elements of the set.

operations: list[d20.SetOperation]
The operations to run on the set.

roll_another()
Rolls another *Die* of the appropriate size and adds it to this set.

class d20.**Die**(*size*, *values*, *context=None*)
Bases: d20.expression.Number

Represents a single die.

size: int
How many sides this Die has.

values: list[*d20.Literal*]
The history of values this die has rolled.

class d20.**SetOperator**(*op*, *sels*)
Represents an operation on a set.

op: str
The operation to run on the selected elements of the set.

sels: list[*d20.SetSelector*]
The selectors that describe how to select operands.

select(*target*, *max_targets=None*)
Selects the operands in a target set.

Parameters

- **target** (Number) – The source of the operands.
- **max_targets** (*Optional[int]*) – The maximum number of targets to select.

operate(*target*)
Operates in place on the values in a target set.

Parameters **target** (Number) – The source of the operands.

class d20.**SetSelector**(*cat*, *num*)
Represents a selection on a set.

cat: Optional[str]
The type of selection (lowest, highest, literal, etc).

num: int
The number to select (the N in lowest/highest/etc N, or literally N).

select(*target*, *max_targets=None*)
Selects operands from a target set.

Parameters

- **target** (Number) – The source of the operands.
- **max_targets** (*int*) – The maximum number of targets to select.

Returns The targets in the set.

Return type set of Number

CHAPTER

FOUR

ABSTRACT SYNTAX TREE

This page documents the structure of the Expression object as returned by `parse(...)`. If you're looking for the Expression object returned by `roll(...)expr`, check out *Expression Tree*.

class d20.ast.**ChildMixin**
A mixin that tree nodes must implement to support tree traversal utilities.

property children
(read-only) The children of this object, usually used for traversing a tree.

Return type list[*Node*]

property left
The leftmost child of this object, usually used for traversing a tree.

Return type *Node*

property right
The rightmost child of this object, usually used for traversing a tree..

Return type *Node*

set_child(*index*, *value*)
Sets the ith child of this object.

Parameters

- **index** (*int*) – The index of the value to set.
- **value** (ChildMixin) – The new value to set it to.

class d20.ast.**Node**
Bases: abc.ABC, d20.diceast.ChildMixin

The base class for all AST nodes.

A Node has no specific attributes, but supports all the methods in *ChildMixin* for traversal.

class d20.ast.**Expression**(*roll*, *comment=None*)
Bases: d20.diceast.Node

Expressions are usually the root of all ASTs.

roll: *d20.ast.Node*
The subtree representing the expression's roll.

comment: Optional[str]
The comment of this expression.

class d20.ast.**AnnotatedNumber**(*value*, **annotations*)
Bases: d20.diceast.Node

Represents a value with an annotation.

value: *d20.ast.Node*
The subtree representing the annotated value.

annotations: list[str]
The annotations on the value.

class d20.ast.**Literal**(*value*)
Bases: d20.diceast.Node

value: Union[int, float]
The literal number represented.

class d20.ast.**Parenthetical**(*value*)
Bases: d20.diceast.Node

value: *d20.ast.Node*
The subtree inside the parentheses.

class d20.ast.**UnOp**(*op*, *value*)
Bases: d20.diceast.Node

op: str
The unary operation.

value: *d20.ast.Node*
The subtree that the operation operates on.

class d20.ast.**BinOp**(*left*, *op*, *right*)
Bases: d20.diceast.Node

op: str
The binary operation.

left: *d20.ast.Node*
The left subtree that the operation operates on.

right: *d20.ast.Node*
The right subtree that the operation operates on.

class d20.ast.**OperatedSet**(*the_set*, **operations*)
Bases: d20.diceast.Node

value: *d20.ast.NumberSet*
The set to be operated on.

operations: list[d20.SetOperation]
The operations to run on the set.

class d20.ast.**NumberSet**(*values*)
Bases: d20.diceast.Node

values: list[*d20.ast.NumberSet*]
The elements of the set.

class d20.ast.**OperatedDice**(*the_dice*, **operations*)
Bases: d20.diceast.OperatedSet

class d20.ast.**Dice**(*num*, *size*)
Bases: d20.diceast.Node

num: int
The number of dice to roll.

size: int
How many sides each die should have.

class d20.ast.**SetOperator**(*op*, *sels*)

op: str
The operation to run on the selected elements of the set.

sels: list[*d20.SetSelector*]
The selectors that describe how to select operands.

class d20.ast.**SetSelector**(*cat*, *num*)

cat: Optional[str]
The type of selection (lowest, highest, literal, etc).

num: int
The number to select (the N in lowest/highest/etc N, or literally N).

CHAPTER

FIVE

UTILITIES

d20.utils.**ast_adv_copy**(*ast: ASTNode*, *advtype: d20.dice.AdvType*) → ASTNode
Returns a minimally shallow copy of a dice AST with respect to advantage.

```
>>> tree = d20.parse("1d20 + 5")
>>> str(tree)
'1d20 + 5'
>>> str(ast_adv_copy(tree, d20.AdvType.ADV))
'2d20kh1 + 5'
```

Parameters

- **ast** (d20.ast.Node) – The parsed AST.
- **advtype** (AdvType) – The advantage type to roll at.

Returns The copied AST.

Return type *d20.ast.Node*

d20.utils.**dfs**(*node: TreeType*, *predicate: Callable[[TreeType], bool]*) → Optional[TreeType]
Returns the first node in the tree such that `predicate(node)` is True, searching depth-first left-to-right. Returns None if no node satisfying the predicate was found.

Parameters

- **node** (d20.ast.ChildMixin) – The root node of the tree.
- **predicate** (*Callable[[d20.ast.ChildMixin], bool]*) – A predicate function.

Return type Optional[*d20.ast.ChildMixin*]

d20.utils.**leftmost**(*root: TreeType*) → TreeType
Returns the leftmost leaf in this tree.

Parameters **root** (d20.ast.ChildMixin) – The root node of the tree.

Return type *d20.ast.ChildMixin*

d20.utils.**rightmost**(*root: TreeType*) → TreeType
Returns the rightmost leaf in this tree.

Parameters **root** (d20.ast.ChildMixin) – The root node of the tree.

Return type *d20.ast.ChildMixin*

d20.utils.**simplify_expr**(*expr: d20.expression.Expression*, ***kwargs*)
Transforms an expression in place by simplifying it (removing all dice and evaluating branches with respect to annotations).

```
>>> roll_expr = d20.roll("1d20[foo] + 3 - 1d4[bar]").expr
>>> simplify_expr(roll_expr)
>>> d20.SimpleStringifier().stringify(roll_expr)
"7 [foo] - 2 [bar] = 5"
```

Parameters

- **expr** (d20.Expression) – The expression to transform.
- **kwargs** – Arguments that are passed to *simplify_expr_annotations()*.

d20.utils.**simplify_expr_annotations**(*expr: ExpressionNode*, *ambig_inherit: Optional[str] = None*)

Transforms an expression in place by simplifying the annotations using a bubble-up method.

```
>>> roll_expr = d20.roll("1d20[foo]+3").expr
>>> simplify_expr_annotations(roll_expr.roll)
>>> d20.SimpleStringifier().stringify(roll_expr)
"1d20 (4) + 3 [foo] = 7"
```

Parameters

- **expr** (d20.Number) – The expression to transform.
- **ambig_inherit** (*Optional[str]*) – When encountering a child node with no annotation and the parent has ambiguous types, which to inherit. Can be `None` for no inherit, `'left'` for leftmost, or `'right'` for rightmost.

d20.utils.**tree_map**(*func: Callable[[TreeType], TreeType]*, *node: TreeType*) → TreeType

Returns a copy of the tree, with each node replaced with `func(node)`.

Parameters

- **func** (*Callable[[d20.ast.ChildMixin], d20.ast.ChildMixin]*) – A transformer function.
- **node** (d20.ast.ChildMixin) – The root of the tree to transform.

Return type *d20.ast.ChildMixin*

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

F

K

L

M

N

O

P

R

S

T

U

V