dice_tables Documentation

Release 2.6.0

Eric Shaw

Dec 05, 2020

CONTENTS

dicetables is a module for getting all the combinations for any set of dice and getting information from that data.

It generates the combinations by adding dice to a DiceTable or DetailedDiceTable. You can get useful strings and calculations with EventsCalulations.

For some quick examples, see *Getting Started*

Contents:

CHAPTER

ONE

BASIC MATH FOR DICE AND EVENTS

1.1 Basics of Dice and combinations

Rolling two six-sided dice provides a good introduction to the math behind die rolls. Below are all the possible ways to roll 2D6 grouped by the total roll:

```
 2: 1-1
 3: 1-2, 2-1
 4: 1-3, 3-1, 2-2
 5: 1-4, 4-1, 2-3, 3-2
 6: 1-5, 5-1, 2-4, 4-2, 3-3
 7: 1-6, 6-1, 2-5, 5-2, 3-4, 4-3
 8: 2-6, 6-2, 3-5, 5-3, 4-4
 9: 3-6, 6-3, 4-5, 5-4
10: 4-6, 6-4, 5-5
11: 5-6, 6-5,
12: 6-6
```

There is only one way to roll a "2", but there are six ways to roll a "7". There are a total of 36 different combinations on these two sets of dice. That means that your chance of rolling a 2 is 1/36 while your chance of rolling a 7 is 6/36.

You could test this with a quick piece of python code, like so:

```
from random import randint

def roll_six():
    return randint(1, 6)

def ten_thousand_rolls_of_2d6():
    answer = {roll: 0 for roll in range(2, 13)}
    for _ in range(10000):
        roll1 = roll_six()
        roll2 = roll_six()
        total= roll1 + roll2
        answer[total] += 1
return answer

print(ten_thousand_rolls_of_2d6())
```

Running this scripts returned the following:

```
{2: 317,
 3: 572,
 4: 813,
```

(continues on next page)

(continued from previous page)

```
 5: 1135,
 6: 1349,
 7: 1635,
 8: 1417,
 9: 1123,
 10: 805,
 11: 548,
 12: 286}
```

Of the 10,000 rolls, 12 was rolled 286 times, or 2.8% (pretty close to 1/36). 7 happened about 16% of the time (pretty close to 6/36).

1.2 Events Dictionary

While the above is great for visualization, it quickly becomes unmanageable. Imagine mapping out all the totals for 3D6 (there are 216 of them), or, even worse, having to keep track of 3 dice of 3 different sizes.

Instead, use dictionary of events having {roll: number_of_times_it_occurs}. On a six-sided die, each roll has an equal chance of occurring, so it's represented by the dictionary: `{1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1}`.

To combine this with another six-sided die, you iterate through each die roll and create a new dictionary. When the second die rolls a "1", it bumps all the rolls up by one, making: `{1+1: 1 ,2+1: 1, 3+1: 1, 4+1: 1, 5+1: 1, 6+1: 1}`. When the second die rolls a "2" it bumps all the die rolls up by two, making: `{1+2: 1 ,2+2: 1, 3+2: 1, 4+2: 1, 5+2: 1, 6+2: 1}`. Keep doing this for each roll and add them up.

```
{2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1}
      {3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 1}
            {4: 1, 5: 1, 6: 1, 7: 1, 8: 1, 9: 1}
                  {5: 1, 6: 1, 7: 1, 8: 1, 9: 1, 10: 1}
                        {6: 1, 7: 1, 8: 1, 9: 1, 10: 1, 11: 1}
+                             {7: 1, 8: 1, 9: 1, 10: 1, 11: 1, 12: 1}
------------------------------------------------------------------------
{2: 1, 3: 2, 4: 3, 5: 4, 6: 5, 7: 6, 8: 5, 9: 4, 10: 3, 11: 2, 12: 1}
```

This dictionary of events is the spread of combinations for 2D6. It is the same as the other representation in the previous section. "2" has one combination, "3" has two combinations and "7" has six combinations.

Using this method, adding another six-sided die to 2D6 becomes much more manageable. Simply take the dictionary of 2D6 above, and for each roll of the next six-sided die, create a new dictionary of events.

When the third six-sided die rolls a "1", you get:

```
{2+1: 1, 3+1: 2, 4+1: 3, 5+1: 4, 6+1: 5, 7+1: 6, 8+1: 5, 9+1: 4, 10+1: 3, 11+1: 2,␣
↪12+1: 1}
```

And when it rolls a "2", you get:

```
{2+2: 1, 3+2: 2, 4+2: 3, 5+2: 4, 6+2: 5, 7+2: 6, 8+2: 5, 9+2: 4, 10+2: 3, 11+2: 2,␣
↪12+2: 1}
```

This makes:

```
{3: 1, 4: 2, 5: 3, 6:  4, 7:  5, 8:  6, 9:  5, 10:  4, 11:  3, 12:  2, 13:  1}
      {4: 1, 5: 2, 6:  3, 7:  4, 8:  5, 9:  6, 10:  5, 11:  4, 12:  3, 13:  2, 14:  1}
```

(continues on next page)

(continued from previous page)

```
            {5: 1, 6:  2, 7:  3, 8:  4, 9:  5, 10:  6, 11:  5, 12:  4, 13:  3, 14:  2,
↪ 15:  1}
                   {6:  1, 7:  2, 8:  3, 9:  4, 10:  5, 11:  6, 12:  5, 13:  4, 14:  3,
↪ 15:  2, 16: 1}
                          {7:  1, 8:  2, 9:  3, 10:  4, 11:  5, 12:  6, 13:  5, 14:  4,
↪ 15:  3, 16: 2, 17: 1}
+                                 {8:  1, 9:  2, 10:  3, 11:  4, 12:  5, 13:  6, 14:  5,
↪ 15:  4, 16: 3, 17: 2, 18: 1}
------------------------------------------------------------------------------------------
↪------------------------------
{3: 1, 4: 3, 5: 6, 6: 10, 7: 15, 8: 21, 9: 25, 10: 27, 11: 27, 12: 25, 13: 21, 14: 15,
↪ 15: 10, 16: 6, 17: 3, 18: 1}
```

Another example with 2D6:

Take 2d6 and add a different die to it. This time, it's a weighted 2-sided die that rolls "2" three times as often as "1". This die is represented by the dictionary: `{1:  1, 2:  3}`.

Two 6-sided dice:

```
{2: 1, 3: 2, 4: 3, 5: 4, 6: 5, 7: 6, 8: 5, 9: 4, 10: 3, 11: 2, 12: 1}
```

A weighted 2-sided die:

```
{1: 1, 2: 3}
```

Applying a roll of "one" to two 6-sided dice:

```
A = {2+1: 1, 3+1: 2, 4+1: 3, 5+1: 4, 6+1: 5, 7+1: 6, 8+1: 5, 9+1: 4, 10+1: 3, 11+1: 2,
↪ 12+1: 1}
```

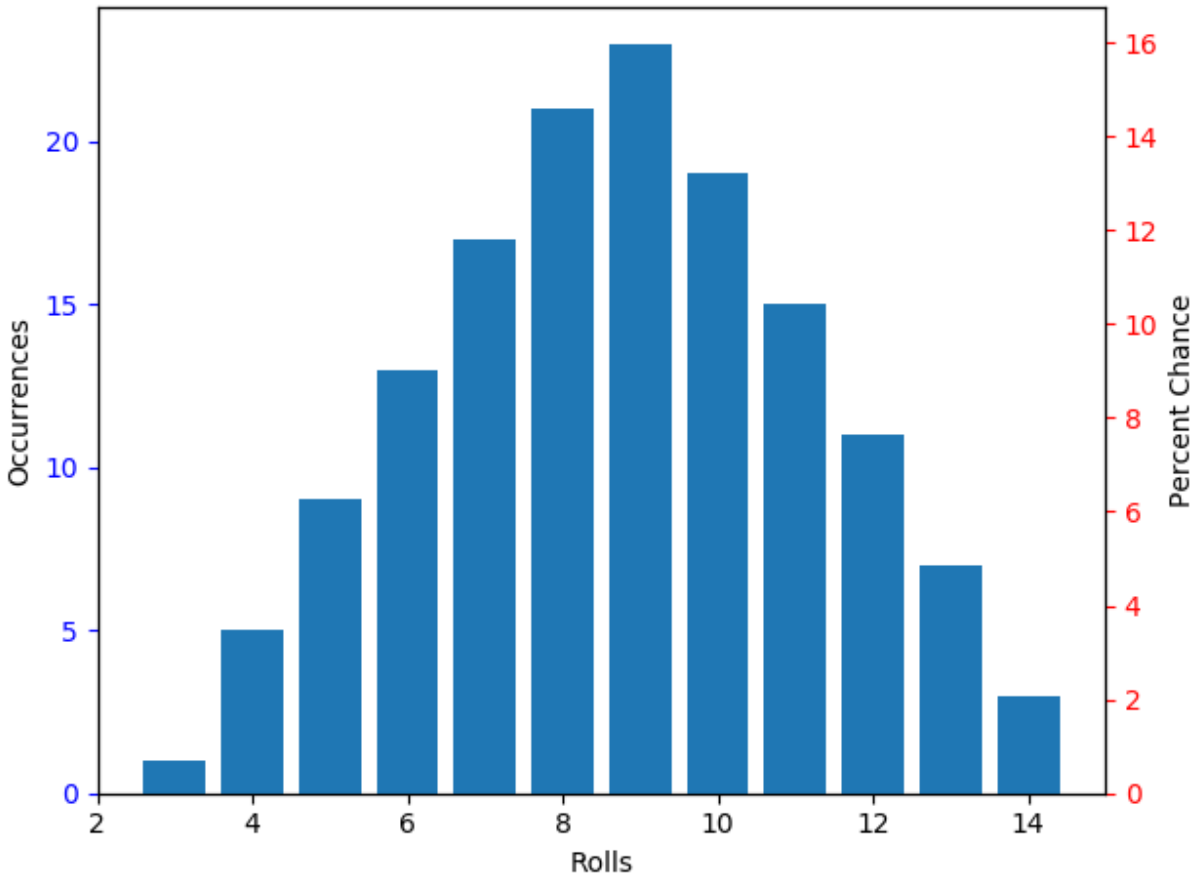
Applying a roll of "two" to two 6-sided dice:

```
B = {2+2: 1, 3+2: 2, 4+2: 3, 5+2: 4, 6+2: 5, 7+2: 6, 8+2: 5, 9+2: 4, 10+2: 3, 11+2: 2,
↪ 12+2: 1}
```

On this weighted die, "2" gets rolled 3 times as often as "1". So you combine *1*A* and *3*B*:

```
{3: 1, 4: 2, 5: 3, 6:  4, 7:  5, 8:  6, 9:  5, 10:  4, 11:  3, 12:  2, 13: 1}
↪# A
      {4: 1, 5: 2, 6:  3, 7:  4, 8:  5, 9:  6, 10:  5, 11:  4, 12:  3, 13: 2, 14: 1}
↪# B
      {4: 1, 5: 2, 6:  3, 7:  4, 8:  5, 9:  6, 10:  5, 11:  4, 12:  3, 13: 2, 14: 1}
↪# B
+     {4: 1, 5: 2, 6:  3, 7:  4, 8:  5, 9:  6, 10:  5, 11:  4, 12:  3, 13: 2, 14: 1}
↪# B
------------------------------------------------------------------------------------------
↪---
{3: 1, 4: 5, 5: 9, 6: 13, 7: 17, 8: 21, 9: 23, 10: 19, 11: 15, 12: 11, 13: 7, 14: 3}
```

So if you rolled 2D6 and 1WeightedDie({1: 1, 2: 3}). You'd get the following distribution. There are 144 total occurrences. 3 happens once, giving it a 0.69% chance, and 9 happens 23 times, giving you a 15.97% chance of rolling a 9.


Occurrences
Percent Chance
Rolls
0
5
10
15
20
0
2
4
6
8
10
12
14
16
2
4
6
8
10
12
14

CHAPTER

TWO

GETTING STARTED

This module has no dependencies and no requirements. So to get started, simply:

```
$ pip install dicetables
```

or:

```
$ git clone https://github.com/eric-s-s/dice-tables.git
$ cd dice-tables
$ python setup.py install
```

The basic objects to use are DiceTable or DetailedDiceTable, and any of the dice classes. They are:

- Die
- ModDie
- WeightedDie
- ModWeightedDie
- Modifier
- StrongDie
- Exploding
- ExplodingOn
- BestOfDicePool
- WorstOfDicePool
- UpperMidOfDicePool
- LowerMidOfDicePool

for details about the dice, see *The Dice*. for details about the dice-tables see *DiceTable and DetailedDiceTable*.

These are all immutable objects. When you add dice to a DiceTable, it returns a new object and doesn't alter the original. Use the `DiceTable.new()` class method to create an empty DiceTable.

```
>>> import dicetables as dt
>>> empty = dt.DiceTable.new()
>>> empty
<DiceTable containing []>
>>> empty.add_die(dt.Die(6), times=10)
<DiceTable containing [10D6]>
>>> empty
<DiceTable containing []>
```

(continues on next page)

(continued from previous page)

```
>>> table = empty.add_die(dt.Die(4), times=3)
>>> table = table.add_die(dt.Die(10), times=5)
>>> table.get_list()
[(Die(4), 3), (Die(10), 5)]
>>> print(table.get_dict())  # This is each roll and how many times it occurs.
{8: 1,
 9: 8,
 10: 36,
 11: 120,
 12: 327,
 ...
 ...
 58: 327,
 59: 120,
 60: 36,
 61: 8,
 62: 1}
```

To get more detailed information, use *EventsCalculations*. It can get the mean, stddev, a nice string of the combinations, points and axes for graphing, and stats for any set of rolls. (and a few others)

```
>>> calculator = dt.EventsCalculations(table)
>>> calculator.mean()
35.0
>>> calculator.stddev(decimal_place=8)
6.70820393
>>> calculator.stats_strings(list(range(8, 20)) + [35] + list(range(50, 63)))
StatsStrings(query_values='8-19, 35, 50-62',
             query_occurrences='515,778',
             total_occurrences='6,400,000',
             one_in_chance='12.41',
             pct_chance='8.059')
>>> calculator.percentage_points()
[(8, 1.5624999999999997e-05),
 (9, 0.00012499999999999998),
 (10, 0.0005625),
 ...
 (59, 0.001875),
 (60, 0.0005625),
 (61, 0.00012499999999999998),
 (62, 1.5624999999999997e-05)]
>>> big_table = dt.DetailedDiceTable.new().add_die(dt.Die(6), 1000)
>>> print(big_table.calc.full_table_string())  # DetailedDiceTable owns an␣
↪EventsCalculations
1000: 1
1001: 1,000
1002: 500,500
1003: 1.672e+8
1004: 4.192e+10
1005: 8.417e+12
...
3513: 1.016e+776
3514: 1.012e+776
3515: 1.007e+776
3516: 1.001e+776
3517: 9.957e+775
```

(continues on next page)

(continued from previous page)

```
3518: 9.898e+775
...
5998: 500,500
5999: 1,000
6000: 1
>>> big_table.calc.stats_strings(list(range(1000, 1501)))
StatsStrings(query_values='1,000-1,500',
             query_occurrences='2.439e+412',
             total_occurrences='1.417e+778',
             one_in_chance='5.809e+365',
             pct_chance='1.722e-364')
```

You can roll events with a *Roller*

```
>>> events = dt.DiceTable.new().add_die(dt.Die(6))
>>> roller = dt.Roller(events)
>>> roller.roll() in [1, 2, 3, 4, 5, 6]
True
```

CHAPTER

THREE

THE DICE

- *Die Classes*
- *Dice Pools*
- *Some Example Dice*

A die class is a *dicetables.eventsbases.protodie.ProtoDie*, which is a subclass of *dicetables.eventsbases.integerevents.IntegerEvents*. It is a representation of die.

All dice require implementations of the following methods:

- **get_dict()** [The representation of the die rolls as `{roll: frequency}`.]

```
>>> import dicetables as dt
>>> dt.Die(3).get_dict() == {1: 1, 2: 1, 3: 1}
True
>>> dt.ModDie(3, -2).get_dict() == {-1: 1, 0: 1, 1: 1}
True
```

- **get_size()** [The size of the die. This can occasionally be non-intuitive.]

```
>>> die = dt.WeightedDie({1: 1, 2: 1, 3: 0})
>>> die.get_size()
3
>>> die.get_dict() == {1: 1, 2: 1}
True
```

- `get_weight()`: The total weight of all the die rolls. Used mainly in the `__lt__` method to differentiate between dice of equal size.
- `weight_info()`: A string detailing the rolls and their weights.
- **multiply_str(number)** [The string representation for multiples of the die.]

```
>>> die = dt.ModDie(6, 1)
>>> str(die)
'D6+1'
>>> die.multiply_str(5)
'5D6+5'
```

- `__str__()`
- `__repr__()`

Dice are immutable , hashable and rich-comparable. Multiple names can safely point to the same instance of a Die, they can be used in sets and dictionary keys and they can be sorted with any other kind of die. Comparisons are done by (size, weight, get_dict, __repr__(as a last resort)). So:

```
>>> dice_list = [
... dt.ModDie(2, 0),
... dt.WeightedDie({1: 1, 2: 1}),
... dt.Die(2),
... dt.ModWeightedDie({1: 1, 2: 1}, 0),
... dt.StrongDie(dt.Die(2), 1),
... dt.StrongDie(dt.WeightedDie({1: 1, 2: 1}), 1)
... ]
>>> [die.get_dict() == {1: 1, 2: 1} for die in dice_list]
[True, True, True, True, True, True]
>>> sorted(dice_list)
[Die(2),
 ModDie(2, 0),
 StrongDie(Die(2), 1),
 ModWeightedDie({1: 1, 2: 1}, 0),
 StrongDie(WeightedDie({1: 1, 2: 1}), 1),
 WeightedDie({1: 1, 2: 1})]
>>> [die == dt.Die(2) for die in sorted(dice_list)]
[True, False, False, False, False, False]
>>> my_set = {dt.Die(6)}
>>> my_set.add(dt.Die(6))
>>> my_set == {dt.Die(6)}
True
>>> my_set.add(dt.ModDie(6, 0))
>>> my_set == {dt.Die(6), dt.ModDie(6, 0)}
True
```

Top

3.1 Die Classes

class dicetables.dieevents.**Die**(*die_size: int*)

stores and returns info for a basic Die. `Die(4)` rolls 1, 2, 3, 4 with equal weight

Base methods for all dice:

get_size()

get_weight()

get_dict() → Dict[int, int]

Returns {event: occurrences}

weight_info()

return detailed info of how the die is weighted

multiply_str(*number*)

return a string that is the die string multiplied by a number. i.e., D6+1 times 3 is '3D6+3'

class dicetables.dieevents.**ModDie**(*die_size: int*, *modifier: int*)

stores and returns info for a Die with a modifier that changes the values of the rolls. `ModDie(4, -1)` rolls 0, 1, 2, 3 with equal weight

It is 4-sided die with -1 added to each roll (D4-1)

added methods:

get_modifier() → int

class dicetables.dieevents.**WeightedDie**(*dictionary_input: Dict[int, int]*)

stores and returns info for die with different chances for different rolls. `WeightedDie({1:1, 2:5})` rolls 1 once for every five times that 2 is rolled.

dt.WeightedDie({1:1, 3:3, 4:6}) is a 4-sided die. It rolls 4 six times as often as 1, rolls 3 three times as often as 1 and never rolls 2

added methods:

get_raw_dict() → Dict[int, int]

get_raw_dict() returns something similar to the input dict with keys from 1 to *die.get_size()* even if they are zero. `dt.WeightedDie({1: 1, 3: 3, 4: 6}).get_raw_dict()` returns `{1: 1, 2: 0, 3: 3, 4: 4}`

class dicetables.dieevents.**ModWeightedDie**(*dictionary_input: Dict[int, int]*, *modifier: int*)

stores and returns info for die with different chances for different rolls. The modifier changes all die rolls. `ModWeightedDie({1:1, 3:5}, -1)` is a 3-sided die - 1. It rolls 0 once for every five times that 2 is rolled.

added methods:

get_modifier() → int

get_raw_dict() → Dict[int, int]

```
>>> dt.WeightedDie({1: 1, 3: 3, 4: 6}).get_raw_dict() == {1: 1, 2: 0, 3: 3, 4: 6}
True
>>> dt.ModWeightedDie({1: 1, 3: 3, 4: 6}, -100).get_raw_dict() == {1: 1, 2: 0, 3: 3,␣
↪4: 6}
True
```

class dicetables.dieevents.**StrongDie**(*input_die:* dicetables.eventsbases.protodie.ProtoDie, *multiplier: int*)

stores and returns info for a stronger version of another die (including StrongDie if you're feeling especially silly). The multiplier multiplies all die rolls of original Die. `StrongDie(ModDie(3, -1), 2)` rolls (1-1)*2, (2-1)*2, (3-1)*2 with equal weight.

```
>>> die = dt.Die(4)
>>> die.get_dict() == {1: 1, 2: 1, 3: 1, 4: 1}
True
>>> dt.StrongDie(die, 5).get_dict() == {5: 1, 10: 1, 15: 1, 20: 1}
True
>>> example = dt.StrongDie(die, -2)
>>> example.get_dict() == {-2: 1, -4: 1, -6: 1, -8: 1}
True
>>> example.get_input_die() == die
True
>>> example.get_multiplier()
-2
```

added methods:

get_multiplier()

get_input_die() → *dicetables.eventsbases.protodie.ProtoDie*

returns an instance of the original die

class dicetables.dieevents.**Modifier**(*modifier: int*)

stores and returns info for a modifier to add to the final die roll. `Modifier(-3)` rolls -3 and only -3. A Modifier's size and weight are always 0.

```
>>> table = dt.DiceTable.new().add_die(dt.Die(4))
>>> table.get_dict() == {1: 1, 2: 1, 3: 1, 4: 1}
True
>>> table = table.add_die(dt.Modifier(3))
>>> print(table)
+3
1D4
>>> table.get_dict() == {4: 1, 5: 1, 6: 1, 7: 1}
True
```

added methods:

get_modifier() → int

class dicetables.dieevents.**Exploding**(*input_die:* dicetables.eventsbases.protodie.ProtoDie, *explosions: int = 2*)

Stores and returns info for an exploding version of another die. Each time the highest number is rolled, you add that to the total and keep rolling. An exploding D6 rolls 1-5 as usual. When it rolls a 6, it re-rolls and adds that 6. If it rolls a 6 again, this continues, adding 12 to the result. Since this is an infinite but increasingly unlikely process, the "explosions" parameter sets the number of re-rolls allowed.

Explosions are applied after modifiers and multipliers. `Exploding(ModDie(4, -2))` explodes on a 2 so it rolls: [-1, 0, 1, (2 -1), (2 + 0), (2 + 1), (2+2 - 1) ..]

WARNING: setting the number of explosions too high can make instantiation VERY slow. The time is proportional to explosions and die_size.

Here are the rolls for an exploding D4 that can explode up to 3 times. It rolls 1-3 sixty-four times more often than 13-16.

```
>>> roll_values = dt.Exploding(dt.Die(4), explosions=3).get_dict()
>>> sorted(roll_values.items())
 [(1, 64), (2, 64), (3, 64),
  (5, 16), (6, 16), (7, 16),
  (9, 4), (10, 4), (11, 4),
  (13, 1), (14, 1), (15, 1), (16, 1)]
```

Any modifiers and multipliers are applied to each re-roll. Exploding D6+1 explodes on a 7. On a "7" it rolls 7 + (D6 + 1). On a "14", it rolls 14 + (D6 + 1).

Here are the rolls for an exploding D6+1 that can explode the default times.

```
>>> roll_values = dt.Exploding(dt.ModDie(6, 1)).get_dict()
>>> sorted(roll_values.items())
[(2, 36), (3, 36), (4, 36), (5, 36), (6, 36),
 (9, 6), (10, 6), (11, 6), (12, 6), (13, 6),
 (16, 1), (17, 1), (18, 1), (19, 1), (20, 1), (21, 1)]
```

added methods:

get_explosions() → int

get_input_die() → *dicetables.eventsbases.protodie.ProtoDie*
returns an instance of the original die

class dicetables.dieevents.**ExplodingOn**(*input_die:* dicetables.eventsbases.protodie.ProtoDie, *explodes_on: Iterable[int]*, *explosions: int = 2*)

Stores and returns info for an exploding version of another die. Each time the values in (explodes_on) are rolled, the die continues to roll, adding that value to the result. The die only continues rolling an (explosions) number of times.

`ExplodingOn(Die(6), (1, 6), explosions=2)` rolls: [2 to 5], 1+[2 to 5], 6+[2 to 5], 1+1+[1 to 6], 1+6+[1 to 6], 6+1+[1 to 6] and 6+6+[1 to 6].

Explosions are applied after modifiers and multipliers. `ExplodingOn(ModDie(4, -2), (2,))` explodes on a 2 so it rolls: [-1, 0, 1, (2 -1), (2 + 0), (2 + 1), (2+2 - 1) ..]

WARNING: setting the number of explosions too high can make instantiation VERY slow. Time is proportional to explosion**(len(explodes_on)). It's also linear with size which gets overshadowed by the first factor.

Here are the rolls for an exploding D6 that can explode the default times and explodes on 5 and 6.

```
>>> roll_values = dt.ExplodingOn(dt.Die(6), (5, 6)).get_dict()
>>> sorted(roll_values.items())
[(1, 36), (2, 36), (3, 36), (4, 36),
 (6, 6), (7, 12), (8, 12), (9, 12), (10, 6),
 (11, 1), (12, 3), (13, 4), (14, 4), (15, 4), (16, 4), (17, 3), (18, 1)]
```

added methods:

get_explosions()

get_explodes_on() → Tuple[int, . . .]

get_input_die() → *dicetables.eventsbases.protodie.ProtoDie*
returns an instance of the original die

Top

3.2 Dice Pools

`DicePool` s are a pool of a single die. `DicePoolCollection` s are lightweight wrappers around a DicePool. They are a way to extract rolls from a Dice Pool and cast it as a `ProtoDie`. `DicePool` can be expensive to instantiate, which is explained below. They are immutable and a single instance can be passed to many collections.

class dicetables.dicepool.**DicePool**(*input_die:* dicetables.eventsbases.protodie.ProtoDie, *pool_size: int*)

property die

property size

property rolls

The collections are treated as one giant Die with very funky rolling behavior. They all follow the basic form: `<WhatToSelect>OfDicePool(pool=DicePool(input_die, pool_size), select=<int>)`. `BestOfDicePool(DicePool(Die(6), 4), 3)` means: Make a dice pool of 4D6. Roll this and take the best three results from every roll. This object is also an 18-sided "Die" that rolls from 3 to 18.

class dicetables.dicepool_collection.**DicePoolCollection**(*pool:* dicetables.dicepool.DicePool, *select: int*)

The abstract class for all DicePoolCollection objects. A DicePoolCollection creates a new die by selecting from a DicePool. Select determines how many rolls are selected from the pool of total rolls. Different implementations determine which particular rolls to select.

get_pool() → *dicetables.dicepool.DicePool*

get_select() → int

class dicetables.dicepool_collection.**BestOfDicePool**(*pool: dicetables.dicepool.DicePool*, *select: int*)

Take the best [select] rolls from a DicePool of [pool_size] * [input_die]. BestOfDicePool(DicePool(Die(6), 4), 3) is the best 3 rolls from four six-sided dice.

class dicetables.dicepool_collection.**WorstOfDicePool**(*pool: dicetables.dicepool.DicePool*, *select: int*)

Take the worst [select] rolls from a DicePool of [pool_size] * [input_die]. WorstOfDicePool(DicePool(Die(6), 4), 3) is the worst 3 rolls from four six-sided dice.

class dicetables.dicepool_collection.**UpperMidOfDicePool**(*pool: dicetables.dicepool.DicePool*, *select: int*)

Take the middle [select] rolls from a DicePool of [pool_size] * [input_die]. UpperMidOfDicePool(DicePool(Die(6), 5), 3) is the middle 3 rolls from five six-sided dice.

If there is no perfect middle, take the higher of two choices. For five dice that roll (1, 1, 2, 3, 4), select=3 takes (1, 2, 3) and select=2 takes (2, 3).

class dicetables.dicepool_collection.**LowerMidOfDicePool**(*pool: dicetables.dicepool.DicePool*, *select: int*)

Take the middle [select] rolls from a DicePool of [pool_size] * [input_die]. LowerMidOfDicePool(DicePool(Die(6), 5), 3) is the middle 3 rolls from five six-sided dice.

If there is no perfect middle, take the lower of two choices. For five dice that roll (1, 1, 2, 3, 4), select=3 takes (1, 2, 3) and select=2 takes (1, 2).

All DicePool objects calculate all the possible combinations of rolls and the frequency of each combination. So, *DicePool(Die(3), 3, 2)* creates the following dictionary

```
>>> pool = dt.DicePool(dt.Die(3), 3)
>>> pool.rolls == {
... (1, 1, 1): 1,
... (1, 1, 2): 3,
... (1, 1, 3): 3,
... (1, 2, 2): 3,
... (1, 2, 3): 6,
... (1, 3, 3): 3,
... (2, 2, 2): 1,
... (2, 2, 3): 3,
... (2, 3, 3): 3,
... (3, 3, 3): 1
... }
True
```

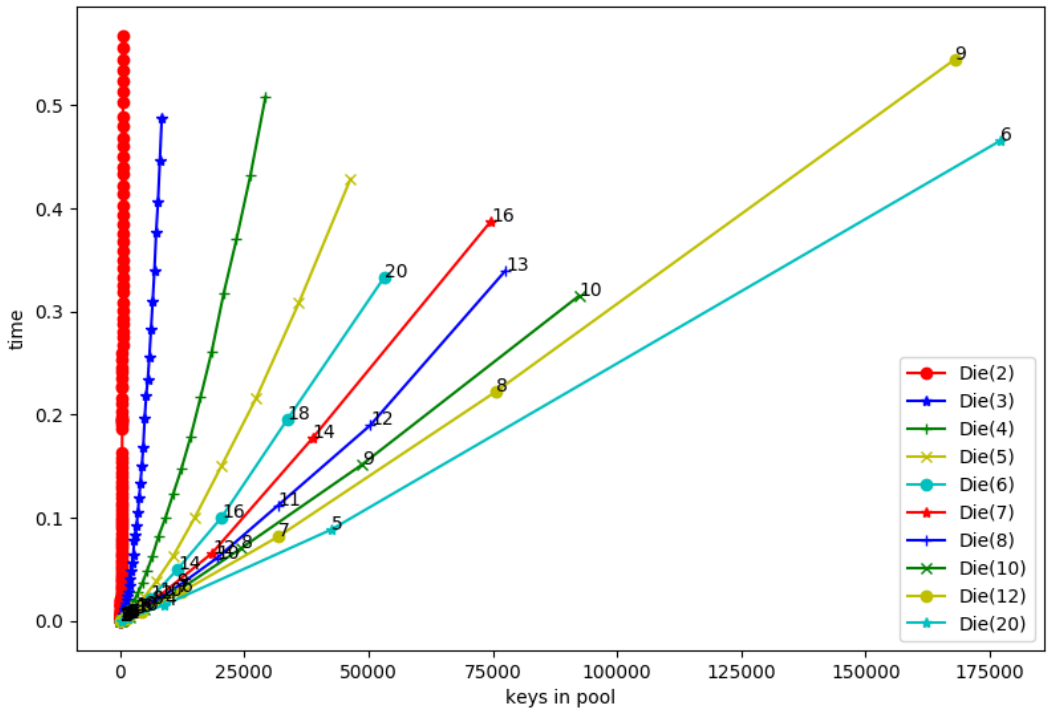
This says that, with 3*Die(3), the roll: (1, 1, 1) happens once. The roll: (1, 2, 3) happens 6 times. `BestOfDicePool(DicePool(Die(3), 3), 2)` looks at the above dictionary and selects the two best rolls in each tuple. so:

```
>>> best_two = dt.BestOfDicePool(pool, 2)
>>> best_two.get_dict() == {2: 1, 3: 3, 4: 7, 5: 9, 6: 7}
True
```

The number of keys in any one of these dictionaries relies on pool_size and `dict_size = len(input_die.get_dict())`. The formula is *(dict_size-1 + pool_size)!/(dict_size-1)! * 1/(pool_size)!* and you can calculate it using *count_unique_combination_keys*. If you have a key_count, you can find the pool_size with *largest_permitted_pool_size*.

```
>>> from dicetables.tools.orderedcombinations import count_unique_combination_keys,␣
↪largest_permitted_pool_size
>>> count_unique_combination_keys(dt.Die(3), 3) == 10  # The dictionary demonstrated␣
↪above
True
>>> count_unique_combination_keys(dt.Die(6), 10) == 3003
True
>>> count_unique_combination_keys(dt.Die(6), 20) == 53130
True
>>> count_unique_combination_keys(dt.Die(6), 30) == 324632
True
>>> largest_permitted_pool_size(dt.Die(6), 330000)
30
```

This graph gives an idea of the times to instantiate different DicePools. It is time vs number-of-keys-needed-to-generate-the-pool. The black annotations are the pool sizes. Notice that each of these increases linearly with the underlying dictionary, but closer to exponentially with pool_size. Especially with larger dice, an increase of one in the pool size can have a surprisingly large effect.



3.3 Some Example Dice

- `WeightedDie({1: 3, 2: 4, 3: 4, 4: 4, 5: 4, 6: 5})` a mildly weighted die that has a 21% chance to roll a "6" (5/24), a 12.5% chance to roll a "1" and the rest are 1 in 6 (4/24).
- `ModWeightedDie({1: 3, 2: 1, 3: 1, 4: 1}, -1)` a six-sided die with faces [0, 0, 0, 1, 2, 3].
- `ModDie(2, -1)` a coin where "1" is heads, and "0" is tails. The die roll will tell you the number of heads rolled.
- `ModWeightedDie({1: 40, 2: 60}, -1)` a cheater's coin that rolls heads 60% of the time.
- `ModWeightedDie({1: 45, 2: 55}. -1)` a person who's likely to pick "1" 55% of the time.
- `StrongDie(ModWeightedDie({1: 10, 2: 90}, -1), 1000)` a thousand people who will almost certainly choose "1" and will all vote as a block. whatever they choose, they're doing it as a team.
- `BestOfDicePool(DicePool(Die(6), 4), 3)` best 3 out of 4D6.
- 2 `Die(6)` and `Modifier(3)` 2D6+3

```
>>> import dicetables as dt
>>> dt.DiceTable.new().add_die(dt.Die(6), 2).add_die(dt.Modifier(3))
<DiceTable containing [+3, 2D6]>
```

Top

CHAPTER

FOUR

DICETABLE AND DETAILEDDICETABLE

The two DiceTable classes are below. They inherit from *dicetables.additiveevents.AdditiveEvents*

class dicetables.dicetable.**DiceTable**(*events_dict: dict*, *dice_record:* dicetables.dicerecord.DiceRecord)

classmethod new() → T

dice_data() → *dicetables.dicerecord.DiceRecord*

get_list()

Returns sorted copy of dice list: [(die, number of dice), . . .]

number_of_dice(*query_die:* dicetables.eventsbases.protodie.ProtoDie) → int

weights_info()

Returns str: complete info for all dice

add_die(*die:* dicetables.eventsbases.protodie.ProtoDie, *times: int = 1*) → T

Parameters

- **die** – any subclass of ProtoDie: Die, ModDie, WeightedDie, ModWeightedDie, Modifier, StrongDie, Exploding, ExplodingOn
- **times** – int>= 0

remove_die(*die:* dicetables.eventsbases.protodie.ProtoDie, *times=1*) → T

Parameters

- **die** – any subclass of ProtoDie: Die, ModDie, WeightedDie, ModWeightedDie, Modifier, StrongDie, Exploding, ExplodingOn
- **times** – 0 <= int <= number of "die" in table

Here is a quick demo of number_of_dice and weights_info

```
>>> import dicetables as dt
>>> table = dt.DiceTable.new().add_die(dt.Die(6), 2).add_die(dt.WeightedDie({1: 2, 3:␣
↪1}))
>>> print(table.weights_info())
1D3  W:3
    a roll of 1 has a weight of 2
    a roll of 2 has a weight of 0
    a roll of 3 has a weight of 1

2D6
    No weights
```

(continues on next page)

(continued from previous page)

```
>>> table.number_of_dice(dt.WeightedDie({1: 2, 2: 0, 3: 1}))
1
>>> table.number_of_dice(dt.Die(6))
2
>>> table.number_of_dice(dt.Die(100))
0
```

class dicetables.dicetable.**DetailedDiceTable**(*events_dict: dict*, *dice_record:* dicetables.dicerecord.DiceRecord, *calc_includes_zeroes: bool = True*)

DetailedDiceTable is a dicetable that owns a *dicetables.eventsinfo.EventsCalculations*. This is accessed from the `.calc` property and you can set whether it includes zero values with *calc_includes_zeros*. Owning an EventsCalculations means that it uses more memory.

property info

property calc

property calc_includes_zeroes

switch_boolean() → T

Returns a new DetailedDiceTable with the *calc_includes_zeroes* boolean switched

You can instantiate any DiceTable or DetailedDiceTable with any data you like. This allows you to create a DiceTable from stored information or to copy. `dice_data` returns the correct input to instantiate a new DiceTable, which is a *dicetables.dicerecord.DiceRecord*(See below for details). To get consistent and sorted output for dice, use `get_list`. Equality testing is by: - type - get_dict()- - dice_data() - (and calc_includes_zeroes for DetailedDiceTable).

```
>>> ten_d_six = dt.DiceTable.new()
>>> ten_d_six = ten_d_six.add_die(dt.Die(6), 10)
>>> events_record = ten_d_six.get_dict()
>>> dice_record = ten_d_six.dice_data()
>>> new_ten_d_six = dt.DiceTable(events_record, dice_record)
>>> print(new_ten_d_six)
10D6
>>> record = dt.DiceRecord({dt.Die(6): 10})
>>> also_ten_d_six = dt.DetailedDiceTable(new_ten_d_six.get_dict(), record, calc_
↪includes_zeroes=False)
>>> ten_d_six.get_dict() == new_ten_d_six.get_dict() == also_ten_d_six.get_dict()
True
>>> ten_d_six.get_list() == new_ten_d_six.get_list() == also_ten_d_six.get_list()
True
>>> ten_d_six == new_ten_d_six
True
>>> ten_d_six == also_ten_d_six  # False by type
False
>>> isinstance(also_ten_d_six, dt.DiceTable)
True
>>> type(also_ten_d_six) is dt.DiceTable
False
```

You can also remove dice, but this will raise an error if you remove too many.

```
>>> table = dt.DiceTable.new().add_die(dt.Die(4), 10)
>>> table.remove_die(dt.Die(4), 7)
<DiceTable containing [3D4]>
```

(continues on next page)

(continued from previous page)

```
>>> table.get_list()
[(Die(4), 10)]
>>> table.remove_die(dt.Die(4), 11)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
DiceRecordError: Tried to create a DiceRecord with a negative value at Die(4): -1
```

DetailedDiceTable.calc_includes_zeroes defaults to True. The only way to change this property is with the `switch_boolean` method, which returns a new DetailedDiceTable that is identical to the original, except with *calc_includes_zeroes* switched.

```
>>> d_table = dt.DetailedDiceTable.new()
>>> d_table.calc_includes_zeroes
True
>>> d_table = d_table.add_die(dt.StrongDie(dt.Die(2), 2))
>>> print(d_table.calc.full_table_string())
2: 1
3: 0
4: 1
```

```
>>> d_table = d_table.switch_boolean()
>>> the_same = dt.DetailedDiceTable({2: 1, 4: 1}, d_table.dice_data(), False)
>>> d_table == the_same
True
>>> print(d_table.calc.full_table_string())
2: 1
4: 1

>>> d_table = d_table.add_die(dt.StrongDie(dt.Die(2), 2))
>>> print(d_table.calc.full_table_string())
4: 1
6: 2
8: 1
```

```
>>> d_table = d_table.switch_boolean()
>>> d_table == the_same
False
>>> print(d_table.calc.full_table_string())
4: 1
5: 0
6: 2
7: 0
8: 1
```

The DiceRecord

class dicetables.dicerecord.**DiceRecord**(*dice_number_dict: Dict[*dicetables.eventsbases.protodie.ProtoDie, *int]*)

This is an immutable record of dice. `add_die`, `remove_die` and `new` return new DiceRecord without altering the original. You can instantiate one with `{dice_object: number_of_dice}`. Trying to create a DiceRecord with a negative number of dice will raise an error.

classmethod new() → *dicetables.dicerecord.DiceRecord*

get_dict() → Dict[*dicetables.eventsbases.protodie.ProtoDie*, int]

get_number(*query_die:* dicetables.eventsbases.protodie.ProtoDie) → int

add_die(*die:* dicetables.eventsbases.protodie.ProtoDie, *times:* *int*) → *dicetables.dicerecord.DiceRecord*

remove_die(*die:* dicetables.eventsbases.protodie.ProtoDie, *times:* *int*) → *dicetables.dicerecord.DiceRecord*

__eq__(*other*)

Return self==value.

CHAPTER

FIVE

EVENTSINFORMATION AND EVENTSCALCULATIONS

These two objects can get information from any Events class. Below the class docs are some code examples.

class dicetables.eventsinfo.**EventsInformation**(*events:* dicetables.eventsbases.integerevents.IntegerEvents)

get_items()

Returns dict.items(): a list in py2 and an iterator in py3.

events_keys() → List[int]

events_range() → Tuple[int, int]

total_occurrences() → int

all_events()

all_events_include_zeroes()

biggest_event() → Tuple[int, int]

Returns (event, occurrences) for first event with highest occurrences

biggest_events_all() → List[Tuple[int, int]]

Returns the list of all events that have biggest occurrence

get_event(*event: int*) → Tuple[int, int]

get_range_of_events(*start: int*, *stop_before: int*) → List[Tuple[int, int]]

class dicetables.eventsinfo.**EventsCalculations**(*events:* dicetables.eventsbases.integerevents.IntegerEvents, *include_zeroes: bool = True*)

property include_zeroes

property info

mean() → float

stddev(*decimal_place=4*) → float

percentage_points() → List[Tuple[int, float]]
Very fast, but only good to ten decimal places.

percentage_points_exact() → List[Tuple[int, float]]

percentage_axes()
Very fast, but only good to ten decimal places.

percentage_axes_exact()

log10_points(*log10_of_zero_value=- 100.0*) → List[Tuple[int, float]]
returns log10 of the occurrences.

Parameters **log10_of_zero_value** – any zero-occurrence must have a preset value.

log10_axes(*log10_of_zero_value=- 100.0*)
returns log10 of the occurrences.

Parameters **log10_of_zero_value** – any zero-occurrence must have a preset value.

full_table_string(*shown_digits=4*, *max_power_for_commaed=6*)

Parameters

- **shown_digits** – How many digits in each scientific notation string.
- **max_power_for_commaed** – The largest power to be represented in comma notation. if set to -1, all numbers are in scientific notation.

stats_strings(*query_list*, *shown_digits=4*, *max_power_for_commaed=6*, *min_power_for_fixed_pt=- 3*)
Calculates the pct chance and one-in chance of any list of numbers, including numbers not in the Events.

Parameters

- **query_list** – A list of ints. Calculates the chance of the list getting rolled.
- **shown_digits** – How many digits in each scientific notation number str.
- **max_power_for_commaed** – The largest power to be represented in comma notation. if set to -1, all numbers >= 1 are in scientific notation.
- **min_power_for_fixed_pt** – The smallest power to be represented in fixed point notation. If set to zero, all values < 1 represented in scientific notation.

Returns (query values, query occurrences, total occurrences, inverse chance, pct chance)

```
>>> import dicetables as dt
>>> table = dt.DiceTable.new().add_die(dt.Die(6), 1000)
>>> calc = dt.EventsCalculations(table)
>>> calc.stddev(7)
54.0061725
>>> calc.mean()
3500.0
>>> the_stats = calc.stats_strings([3500], shown_digits=6) # Shown_digits defaults to␣
↪4.
>>> the_stats
StatsStrings(query_values='3,500',
             query_occurrences='1.04628e+776',
             total_occurrences='1.41661e+778',
             one_in_chance='135.395',
             pct_chance='0.738580')
```

This is correct. Out of 5000 possible rolls, 3500 has a 0.7% chance of occurring.

```
>>> the_stats.one_in_chance
'135.395'
>>> calc.stats_strings(list(range(1000, 3001)) + list(range(4000, 10000)))
StatsStrings(query_values='1,000-3,000, 4,000-9,999',
             query_occurrences='2.183e+758',
             total_occurrences='1.417e+778',
             one_in_chance='6.490e+19',
             pct_chance='1.541e-18')
```

This is also correct. Rolls not in the middle 1000 collectively have a much smaller chance than the mean.

```
>>> silly_table = dt.AdditiveEvents({1: 123456, 100: 1234567*10**1000})
>>> silly_calc = dt.EventsCalculations(silly_table, include_zeroes=False)
>>> print(silly_calc.full_table_string(shown_digits=6))
  1: 123,456
100: 1.23457e+1006
```

EventsCalculations.include_zeroes is only settable at instantiation. It does exactly what it says. EventCalculations owns an EventsInformation. So instantiating EventsCalculations gets you two for the price of one. It's accessed with the property EventsCalculations.info .

```
>>> table = dt.DiceTable.new().add_die(dt.StrongDie(dt.Die(3), 2))
>>> calc = dt.EventsCalculations(table, True)
>>> print(calc.full_table_string())
2: 1
3: 0
4: 1
5: 0
6: 1

>>> calc = dt.EventsCalculations(table, False)
>>> print(calc.full_table_string())
2: 1
4: 1
6: 1

>>> calc.info.events_range()
(2, 6)
```

You can also access some functionality as wrapper functions.

dicetables.eventsinfo.**events_range**(*events*)

dicetables.eventsinfo.**mean**(*events*)

dicetables.eventsinfo.**stddev**(*events*, *decimal_place=4*)

dicetables.eventsinfo.**percentage_points**(*events*, *include_zeroes=True*)

dicetables.eventsinfo.**percentage_axes**(*events*, *include_zeroes=True*)

dicetables.eventsinfo.**stats**(*events*, *query_values*, *shown_digits=4*)

dicetables.eventsinfo.**full_table_string**(*events*, *include_zeroes=True*, *shown_digits=4*)

CHAPTER

SIX

ROLLER

A Roller performs random rolls on any *IntegerEvents*, including dice and DiceTables. You can pass it your own random generator or let it use the python default random generator. Python has specs on how to subclass *random.Random* here.

> Class Random can also be subclassed if you want to use a different basic generator of your own devising: in that case, override the random(), seed(), getstate(), and setstate() methods. Optionally, a new generator can supply a getrandbits() method — this allows randrange() to produce selections over an arbitrarily large range.

This roller relies on *random.randrange* and alias tables. Since it relies on random integer generation and not a random float, it is accurate for any IntegerEvents. So, for instance, on 1000D6, a roll of "1000" has a 1 in 6**1000 chance of occurring (a 7.059e-777% chance). The roller can accurately model that, so that:

```
table = dt.DiceTable.new().add_die(dt.Die(6), 1000)
roller = Roller(table)
my_roll = roller.roll()
```

my_roll actually has a chance of being *1000*, although if that actually happened . . . er . . . um . . . hmmm . . . get your computer checked. Seriously! You probably have a higher chance of winning the lottery while getting eaten by a shark as you're struck by lightning.

class dicetables.roller.**Roller**(*events*, *random_generator: Optional[random.Random] = None*)

property alias_table

here is a nice explanation of alias tables:

Returns dicetables.tools.AliasTable

property random_generator

roll() → int

roll_many(*times*) → List[int]

Parameters **times** – int - how many times to roll

Returns [int,..] - the value of each roll

Here's an example with a custom "random" generator

```
>>> import random
>>> import dicetables as dt
>>> class ZeroForever(random.Random):
...     def __init__(self, *args, **kwargs):
...         self.even_odd_counter = 0
...         super(ZeroForever, self).__init__(*args, **kwargs)
...
```

(continues on next page)

(continued from previous page)

```
...     def randrange(self, start, stop=None, step=1, _int=int):
...         return 0
...
>>> my_die = dt.WeightedDie({1: 1, 2: 10**1000})
>>> roller = dt.Roller(my_die, random_generator=ZeroForever())
>>> roller.roll_many(10)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

CHAPTER

SEVEN

IMPLEMENTATION DETAILS

The following pages give details about.

- events objects: The basis for dice and dicetables: especially IntegerEvents, AdditiveEvents and ProtoDie
- inheritance for dice and dice-tables: There's a little work involved in creating your own objects
- the dice-parser: All its secrets. How to add to it.
- how to get errors and bugs: Details of how to avoid expected problems.

Contents:

7.1 Events Objects

class dicetables.eventsbases.integerevents.**IntegerEvents**

All tables and dice inherit from dicetables.eventsbases.IntegerEvents. IntegerEvents is a collection of {event: number of times it occurs} where events are.... integers! and number of occurrences is int >=0. Any Events is assumed to contain all zero-occurrences events.

All subclasses of IntegerEvents need the method get_dict() which returns {event: occurrences, ...} for each NON-ZERO occurrence. When you instantiate any subclass, it checks to make sure you're get_dict() is legal.

Any of the classes that take a dictionary of events as input scrub the zero occurrences out of the dictionary for you.

```
>>> import dicetables as dt
>>> dt.DiceTable({1: 1, 2: 0}, dt.DiceRecord.new()).get_dict()
{1: 1}
>>> dt.AdditiveEvents({1: 2, 3: 0, 4: 1}).get_dict()
{1: 2, 4: 1}
>>> dt.ModWeightedDie({1: 2, 3: 0, 4: 1}, -5).get_dict()
{-4: 2, -1: 1}
```

Any child of IntegerEvents has access to __eq__ and __ne__ evaluated by type and then get_dict(). It can be compared to any object and two events that are not the exact same class will be !=.

class dicetables.eventsbases.protodie.**ProtoDie**

This is the basis for all the dice classes.

all Die objects need:

- get_size() - returns: int > 0
- get_weight() - returns: int >= 0
- weight_info() - returns: str

- multiply_str() - returns: str
- get_dict() - returns: {int: int > 0}
- __repr__
- __str__

class dicetables.additiveevents.**AdditiveEvents**(*events_dict*)

This is the basis for the *dicetables.dicetable.DiceTable* class. It is an Events that can combine with other Events. It is immutable. When it combines with other Events, the result is a new object.

It has the class method `new()` which returns the identity. This method is inherited by its children. You can add and remove events using the `.combine` method which tries to pick the fastest combining algorithm. You can pick it yourself by calling `.combine_by_<algorithm>`. You can combine and remove DiceTable, AdditiveEvents, Die or any other IntegerEvents, but there's no record of it.

```
>>> three_D2 = dt.AdditiveEvents.new().combine_by_dictionary(dt.Die(2), 3)
>>> also_three_D2 = dt.AdditiveEvents({3: 1, 4: 3, 5: 3, 6: 1})
>>> still_three_D2 = dt.AdditiveEvents.new().combine(dt.AdditiveEvents({1: 1, 2:␣
↪1}), 3)
>>> three_D2.get_dict() == also_three_D2.get_dict() == still_three_D2.get_dict()
True
>>> identity = three_D2.remove(dt.Die(2), 3)
>>> three_D2 == also_three_D2 == still_three_D2
True
>>> identity.get_dict() == dt.AdditiveEvents.new().get_dict() == {0: 1}
True
>>> identity == dt.AdditiveEvents.new()
True
>>> print(three_D2)
table from 3 to 6
>>> twenty_one_D2 = three_D2.combine_by_indexed_values(three_D2, 6)
>>> twenty_one_D2_five_D4 = twenty_one_D2.combine_by_flattened_list(dt.Die(4), 5)
>>> five_D4 = twenty_one_D2_five_D4.remove(dt.Die(2), 21)
>>> dt.DiceTable.new().add_die(dt.Die(4), 5).get_dict() == five_D4.get_dict()
True
>>> dt.DiceTable.new().add_die(dt.Die(4), 5) == five_D4  # will be False since␣
↪DiceTable is not AdditiveEvents
False
```

Since *dicetables.dicetable.DiceTable* is the child of AdditiveEvents, it can do all this combining and removing, but it won't be recorded in the dice record.

classmethod new() → T

get_dict() → Dict[int, int]

Returns {event: occurrences}

combine(*events:* dicetables.eventsbases.integerevents.IntegerEvents, *times: int = 1*) → T

combine_by_flattened_list(*events:* dicetables.eventsbases.integerevents.IntegerEvents, *times: int = 1*) → T

WARNING - UNSAFE METHOD len(flattened_list) = total occurrences of events. if this list is too big, it will raise MemoryError or OverflowError

combine_by_dictionary(*events:* dicetables.eventsbases.integerevents.IntegerEvents, *times: int = 1*) → T

combine_by_indexed_values(*events:* dicetables.eventsbases.integerevents.IntegerEvents, *times: int = 1*) → T

remove (*events:* dicetables.eventsbases.integerevents.IntegerEvents, *times: int = 1*) → T

WARNING - UNSAFE METHOD There is no record of what you added to an AdditiveEvents. If you remove what you haven't added, no error will be raised, but you will have bugs.

7.2 Inheritance

If you inherit from any child of AdditiveEvents and you do not load the new information into EventsFactory, it will complain and give you instructions. The EventsFactory will try to create your new class and if it fails, will return the closest related type

```
>>> import  dicetables as dt
>>> class A(dt.DiceTable):
...     pass
...
>>> A.new()  # EventsFactory takes a stab at it, and guesses right. It returns the␣
↪new class
<...A...>
```

But it also issues a warning:

```
E:\work\dice_tables\dicetables\baseevents.py:74: EventsFactoryWarning:
factory: <class 'dicetables.factory.eventsfactory.EventsFactory'>
Warning code: CONSTRUCT
Failed to find/add the following class to the EventsFactory -
class: <class '__main__.A'>
..... blah blah blah.....
```

Here, it will fail create "B" class, and return its parent.

```
>>> class B(dt.DiceTable):
...     def __init__(self, name, number, events_dict, dice_data):
...         self.name = name
...         self.num = number
...         super(B, self).__init__(events_dict, dice_data)
...
```

```
>>> B.new()
<...DiceTable...>
```

and give you the following warning:

```
E:\work\dice_tables\dicetables\baseevents.py:74: EventsFactoryWarning:
factory: <class 'dicetables.factory.eventsfactory.EventsFactory'>
Warning code: CONSTRUCT
Failed to find/add the following class to the EventsFactory -
class: <class '__main__.B'>
..... blah blah blah.....
```

Now I will try again, but I will give the factory the info it needs. The factory knows how to get 'get_dict', 'dice_data' and 'calc_includes_zeroes'. If you need it to get anything else, you need tuples of (<getter name>, <default value>, 'property' or 'method')

```
>>> class B(dt.DiceTable):
...     factory_keys = ('name', 'get_num', 'get_dict', 'dice_data')
```

(continues on next page)

(continued from previous page)

```
...     new_keys = (('name', '', 'property'), ('get_num', 0, 'method'))
...     def __init__(self, name, number, events_dict, dice_data):
...         self.name = name
...         self._num = number
...         super(B, self).__init__(events_dict, dice_data)
...     def get_num(self):
...         return self._num
...
>>> B.new()
<...B...>
```

```
>>> class C(dt.DiceTable):
...     factory_keys = ('get_dict', 'dice_data')
...     def fancy_add_die(self, die, times):
...         new = self.add_die(die, times)
...         return 'so fancy', new
...
>>> x = C.new().fancy_add_die(dt.Die(3), 2)
>>> x[1].get_dict()
{2: 1, 3: 2, 4: 3, 5: 2, 6: 1}
>>> x
('so fancy', <C...>)
```

Notice that C is returned and not DiceTable

The other way to do this is to directly add the class to the EventsFactory

```
>>> factory = dt.factory.eventsfactory.EventsFactory
>>> factory.add_getter('get_num', 0, 'method')
>>> class A(dt.DiceTable):
...     def __init__(self, number, events_dict, dice):
...         self._num = number
...         super(A, self).__init__(events_dict, dice)
...     def get_num(self):
...         return self._num
...
>>> factory.add_class(A, ('get_num', 'get_dict', 'dice_data'))
>>> A.new()
<A ...>
```

```
>>> factory.reset()
>>> factory.has_class(A)
False
```

When creating new methods, you can generate new events dictionaries by using dicetables.additiveevents.EventsDictCreator. the factory can create new instances with EventsFactory.from_params. For an example see tests.factory.test_eventsfactory 1 . other examples of inheritance with DiceTable can be found just above that at: tests.factory.test_eventsfactory 2

7.3 Parser

- *Customizing Parser*
- *Limiting Max Values*
- *Limits and DicePool Objects*

class dicetables.parser.**Parser**(*ignore_case: bool = False, checker: dicetables.tools.limit_checker.AbstractLimitChecker = <dicetables.tools.limit_checker.NoOpLimitChecker object>*)

__init__(*ignore_case: bool = False, checker: dicetables.tools.limit_checker.AbstractLimitChecker = <dicetables.tools.limit_checker.NoOpLimitChecker object>*)

Parameters

- **ignore_case** – False: Can the parser ignore case on die names and kwargs.
- **checker** – *dicetables.tools.limit_checker.NoOpLimitChecker*: How limits will be enforced for parsing. This defaults to a limit checker that does not check limits.

The Parser object converts strings into dice objects.

```
>>> import dicetables as dt
>>> new_die = dt.Parser().parse_die('Die(6)')
>>> new_die == dt.Die(6)
True
>>> dt.Parser().parse_die('ModDie(6, modifier=1)')
ModDie(6, 1)
```

It can ignore case or not. This applies to dice names and kwarg names. It defaults to ignore_case=False.

```
>>> dt.Parser().parse_die('die(6)')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ParseError: Die class: <die> not recognized by parser.
```

```
>>> dt.Parser(ignore_case=True).parse_die('stronGdie(dIE(6), MULTIPLIER=4)')
StrongDie(Die(6), 4)
```

The Parser can parse all dice in the library: Die, ModDie, WeightedDie, ModWeightedDie, Modifier, StrongDie, Exploding, ExplodingOn and all of the DicePools. It is possible to add other dice to an instance of Parser or make a new class that can parse other dice.

classmethod with_limits(*ignore_case: bool = False, max_size: int = 500, max_explosions: int = 10, max_dice: int = 6, max_dice_pools: int = 2*) → *dicetables.parser.Parser*

Creates a parser with a functioning limit checker from *dicetables.tools.limit_checker.LimitChecker* For explanation of how or why to change *max_dice_pool_combinations_per_dict_size*, and *max_dice_pool_calls*, see Parser

Parameters

- **ignore_case** – False: Can the parser ignore case on die names and kwargs.
- **max_size** – 500: The maximum allowed die size when `parse_die_within_limits`
- **max_explosions** – 10: The maximum allowed (explosions + len(explodes_on)) when `parse_die_within_limits`

- **max_dice** – 6: The maximum number of dice calls when `parse`. Ex: `StrongDie(Exploding(Die(5), 2), 3)` has 3 dice calls.
- **max_dice_pools** – 2: The maximum number of allowed dice_pool calls

property param_types

property classes

parse_die(*die_string*)

make_die(*call_node: _ast.Call*)

make_pool(*call_node: _ast.Call*)

walk_dice_calls(*call_node: _ast.AST*) → Iterable[Union[Type[*dicetables.eventsbases.protodie.ProtoDie*], Type[*dicetables.dicepool.DicePool*]]]

add_class(*class_: object*)

Parameters **class** – the class you are adding

add_param_type(*param_type*, *creation_method*)

7.3.1 Customizing Parser

Parser can only parse very specific types of parameters.

```
>>> from dicetables.parser import make_int, make_int_dict, make_int_tuple
>>> from typing import Dict, Iterable
>>> from dicetables.eventsbases.protodie import ProtoDie
>>> parser = dt.Parser()
>>> parser.param_types == {int: make_int, Dict[int, int]: make_int_dict, dt.DicePool:␣
↪parser.make_pool,
...                        ProtoDie: parser.make_die, Iterable[int]: make_int_tuple}
True
```

If, for example, you need Parser to know how to parse a string, a list of strings and dictionary of keys=str: values=int, you first need to create functions that can parse the appropriate Nodes. Then you assign the functions to the parser.

First, a very very quick introduction to the Abstract Syntax Tree:

The nodes are derived using the ast module. ast, very briefly, takes a string and parses it into nodes. To see what it does, use `ast.dump(ast.parse(<your_string>))`. Create and test nodes by using `my_node = ast.parse(<your_string>).body[0].value`

Note that before py3.8 the elipsis below will be classes "Num" and "Str". After 3.8 they are "Constant".

```
>>> import ast
>>> ast.dump(ast.parse('{1: "a", 2: "b"}'))
"Module(body=[Expr(value=Dict(keys=[...], values=[...]))]...)"
>>> my_list_node = ast.parse('[1, "A"]').body[0].value
>>> ast.dump(my_list_node)
"List(elts=[...], ctx=Load())"
```

This says that the List node points to its elts:

- a Num node: value=1
- a Str node: value='A'

Now, to my example.

```
>>> str_value =  ast.Str(s="abd")
>>> str_value.s
'abd'
>>> str_list = ast.List(elts=[ast.Str(s='a'), ast.Str(s='b'), ast.Str(s='c')])
>>> str_int_dict = ast.parse("{'a': 2, 'b': 10}").body[0].value
```

and here are conversion methods.

```
>>> from dicetables.parser import make_int
>>> def make_str(str_node):
...     return str_node.s
>>> make_str(str_value)
'abd'
```

```
>>> def make_str_list(lst_node):
...     return [make_str(node) for node in lst_node.elts]
>>> make_str_list(str_list)
['a', 'b', 'c']
```

```
>>> def make_str_int_dict(dict_node):
...     keys = [make_str(node) for node in dict_node.keys]
...     values = [make_int(node) for node in dict_node.values]
...     return dict(zip(keys, values))
>>> make_str_int_dict(str_int_dict) == {'a': 2, 'b': 10}
True
```

Now you tell the parser that a key of your choice corresponds to the method. Notice that we are specifically setting the typing to List and not Iterable. If we also want to be able to handle *Iterable[str]*, we will need to explicitly add that. Our use-case takes a list and copies it. That requires a *List*. But the case where we just need an iterable would be different.

```
>>> from typing import List
>>> parser = dt.Parser()
>>> parser.add_param_type(str, make_str)
>>> parser.add_param_type(List[str], make_str_list)
>>> parser.add_param_type(Dict[str, int], make_str_int_dict)
```

To add a new dice class to the parser, give the parser the class and a tuple of the param_types keys for each parameter. The parser will assume you're adding a class with an __init__ function and will try to auto_detect kwargs. You can disable this and add your own kwargs (or not).

To add a new dice class to the parser, **you must use type hints**. The parser uses the annotations in the *__init__* to parse a die

```
>>> class NamedDie(dt.Die):
...     def __init__(self, name: str, buddys_names: List[str], stats: Dict[str, int],␣
↪size: int):
...         self.name = name
...         self.best_buds = buddys_names[:]
...         self.stats = stats.copy()
...         super(NamedDie, self).__init__(size)
...
...     def __eq__(self, other):
...         return (super(NamedDie, self).__eq__(other) and
...                 self.name == other.name and
...                 self.best_buds == other.best_buds and
...                 self.stats == other.stats)
```

```
>>> parser.add_class(NamedDie)
>>> die_str = 'NamedDie("Tom", ["Dick", "Harry"], stats={"friends": 2, "coolness_
↪factor": 10}, size=4)'
>>> die = NamedDie('Tom', ['Dick', 'Harry'], {'friends': 2, 'coolness_factor': 10}, 4)
>>> parser.parse_die(die_str) == die
True
```

You can make a new parser class instead of a specific instance of Parser.

```
>>> class MyParser(dt.Parser):
...     def __init__(self, ignore_case=False):
...         super(MyParser, self).__init__(ignore_case)
...         self.add_param_type(str, make_str)
...         self.add_param_type(List[str], make_str_list)
...         self.add_param_type(Dict[str, int], make_str_int_dict)
...         self.add_class(NamedDie)
```

```
>>> die_str = 'NamedDie("Tom", ["Dick", "Harry"], {"friends": 2, "coolness_factor":␣
↪10}, 4)'
>>> t_d_and_h_4_eva = NamedDie('Tom', ['Dick', 'Harry'], {'friends': 2, 'coolness_
↪factor': 10}, 4)
>>> MyParser().parse_die(die_str) == t_d_and_h_4_eva
True
>>> upper_lower_who_cares = 'nAmeDdIE("Tom", ["Dick", "Harry"], stats={"friends": 2,
↪"coolness_factor": 10}, size=4)'
>>> MyParser(ignore_case=True).parse_die(upper_lower_who_cares) == t_d_and_h_4_eva
True
```

7.3.2 Limiting Max Values

A parser is instantiated with a *dicetables.tools.limit_checker.AbstractLimitChecker*.

class dicetables.tools.limit_checker.**AbstractLimitChecker**

abstract assert_numbers_of_calls_within_limits(*die_classes: Iterable[Union[Type[*dicetables.eventsbases.protodie.ProtoDie*]], Type[*dicetables.dicepool.DicePool*]]]*) → None

asserts that the number of *dicetables.ProtoDie* calls and the number of *dicetables.dicepool.DicePool* calls are within limits.

Raises **LimitsError** –

abstract assert_die_size_within_limits(*bound_args: inspect.BoundArguments*) → None

Checks the bound arguments for the size of the die and asserts they are within limits.

This typically uses *dicetables.tools.limit_checker.get_bound_args* to determine what is a *dicetables.tools.limit_checker.ArgumentType.SIZE* argument.

Raises **LimitsError** –

abstract assert_explosions_within_limits(*bound_args: inspect.BoundArguments*) → None

Asserts that the number of explosions on an *dicetables.Exploding* or an *dicetables.ExplodingOn* has a number of explosions withing limits.

This typically uses *dicetables.tools.limit_checker.get_bound_args* to determine what is a *dicetables.tools.limit_checker.ArgumentType.EXPLOSIONS* and a *dicetables.tools.limit_checker.ArgumentType.EXPLODES_ON* argument.

Raises **LimitsError** –

abstract assert_dice_pool_within_limits(*bound_args: inspect.BoundArguments*) → None

asserts that the size of a dice pool is within limits. Dice pools can be very expensive to calculate. see:

DicePools

This typically uses *dicetables.tools.limit_checker.get_bound_args* to determine what is a *dicetables.tools.limit_checker.ArgumentType.POOL_SIZE* argument.

Raises **LimitsError** –

The default limit checker is a *NoOpLimitChecker* which does not check limits. If you use the factory function, *Parser.with_limits()*, you will create a parser that uses a `dicetables.tools.limit_checker.LimitChecker`.

The size is limited according to the *die_size* parameter or the max value of the *dictionary_input* parameter. The explosions is limited according to *explosions* parameter and the *len* of the *explodes_on* parameter. The number of nested dice is limited according to how many times the parser has to make a die while creating the die. `StrongDie(Exploding(Die(4)), 3)` has three calls.

The number of nested dice is calculated according to how many times *Parser.make_die()* is called. In order to check the size and explosions, the parser must know what parameter name is assigned to values that control size and explosions. It recognizes the following kwarg names:

- 'die_size'
- 'dictionary_input'
- 'explosions'
- 'explodes_on'
- 'input_die'
- 'pool_size'

If you make a die that doesn't use these key-word arguments, the parser will have no way to check limits for you and will simply parse the die string.

ex:

```
>>> parser = dt.Parser.with_limits()
>>> parser.parse_die('Die(500)')
Die(500)
>>> parser.parse_die('Die(501)')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LimitsError: Max die_size: 500
>>> stupid = 'StrongDie(' * 20 + 'Die(5)' + ', 2)' * 20
>>> parser.parse_die(stupid)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LimitsError: Limits exceeded. Max dice calls: 6.
```

```
>>> class NewDie(dt.Die):
...     def __init__(self, funky_new_die_size: int = 6):
...         super(NewDie, self).__init__(funky_new_die_size)
```

(continues on next page)

(continued from previous page)

```
...
...     def __repr__(self):
...         return 'NewDie({})'.format(self.get_size())
```

```
>>> parser = dt.Parser.with_limits()
>>> parser.add_class(NewDie)
```

```
>>> parser.parse_die('NewDie(5000)')
NewDie(5000)
>>> parser.parse_die('Die(5000)')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LimitsError: Limits exceeded. Max dice calls: 6.
```

In order to make sure your new and exciting key-word gets checked, you'll need to subclass the Limit Checker

```
>>> from dicetables.tools.limit_checker import LimitChecker, LimitsError
>>> from inspect import BoundArguments
>>> def get_new_size_args(bound_args):
...     return bound_args.arguments.get("funky_new_die_size")
```

```
>>> class MyChecker(LimitChecker):
...     def assert_die_size_within_limits(self, bound_args: BoundArguments) -> None:
...         super(MyChecker, self).assert_die_size_within_limits(bound_args)
...         new_size_arg = get_new_size_args(bound_args)
...         if new_size_arg and new_size_arg > self.max_size:
...             raise LimitsError("your funky new die size is too funky fresh for␣
↪this parser")
```

```
>>> new_parser = dt.Parser(checker=MyChecker())
>>> new_parser.add_class(NewDie)
```

```
>>> new_parser.parse_die('NewDie(5000)')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LimitsError: your funky new die size ...
>>> new_parser.parse_die('Die(5000)')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LimitsError: Limits exceeded. Max dice calls: 6.
```

7.3.3 Limits and DicePool Objects

DicePool can take a surprisingly long time to calculate. See the *Dice Pools* for a proper explanation. Suffice it to say that the limits on any DicePool can be determined by `len(input_die.get_dict())` and `pool_size`. The parser uses a dictionary of {max_dict_len: max_unique_combination_keys} at `Parser().max_dice_pool_combinations_per_dict_size`. This is determined from the input_die using, `dicetables.tools.orderedcombinations.count_unique_combination_keys(events, pool_size)()`. The current dictionary was determined using the extremely scientific approach of "trying different things and seeing how long they took". This is likely going to be different with whatever computer you will be using. That's why this a public variable.

The other variable is `Parser.with_limits().checker.max_dice_pools`, currently set to "2". This is

separate from *max_nested_dice*. This checks the number of calls to construct a `Dicepool` The sheer unreadability of nested dice pools does beg the question of why you would ever want to do this.

```
>>> parser = dt.Parser.with_limits(max_dice=4)
>>> two_pools_three_dice =
↪'BestOfDicePool(DicePool(StrongDie(WorstOfDicePool(DicePool(Die(2), 3), 2), 2), 3),␣
↪2)'
>>> parser.parse_die(two_pools_three_dice)
BestOfDicePool(DicePool(StrongDie(WorstOfDicePool(DicePool(Die(2), 3), 2), 2), 3), 2)
>>> three_pools =
↪'BestOfDicePool(DicePool(BestOfDicePool(DicePool(WorstOfDicePool(DicePool(Die(2),␣
↪3), 2), 3), 2), 4), 3)'
>>> parser.parse_die(three_pools)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LimitsError: "Limits exceeded. Max dice calls: 4. Max dice pool calls: 2 ...
>>> five_dice = 'BestOfDicePool(DicePool(StrongDie(StrongDie(StrongDie(Die(2), 2), 2),
↪ 2), 2), 2)'
>>> parser.parse_die(five_dice)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LimitsError: "Limits exceeded. Max dice calls: 4. Max dice pool calls: 2 ...
```

With the current limits in place, an implementation of a dice pool could take up to 0.5s. If five calls were allowed, that would be 2.5s to parse a single die. It is hard to imagine any practical reason to use more than one pool. `BestOfDicePool(DicePool(WorstOfDicePool(DicePool(Die(6), 4), 3), 2), 1)` would mean: "Roll 4D6 and take the worst three. Do that twice and take the best one". If the current limit of two feels too limiting, change it.

7.4 How to Get Errors and Bugs

- *get_dict errors*
- *errors for dice*
- *add_die and remove_die are relatively safe*
- *combine and remove are not*
- *making a DiceTable with nonsense*

7.4.1 get_dict errors

Every time you instantiate any IntegerEvents, it is checked. The get_dict() method returns a dict, and every value in get_dict().values() must be >=1. get_dict() may not be empty. since dt.Die(-2).get_dict() returns {}

```
>>> import dicetables as dt
>>> dt.Die(-2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
InvalidEventsError: events may not be empty. a good alternative is the identity - {0:␣
↪1}.
```

```
>>> dt.AdditiveEvents({1.0: 2})
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
InvalidEventsError: all values must be ints
```

```
>>> dt.WeightedDie({1: 1, 2: -5})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
InvalidEventsError: no negative or zero occurrences in Events.get_dict()
```

Because AdditiveEvents and WeightedDie specifically scrub the zeroes from their get_dict() methods, these will not throw errors.

```
>>> dt.AdditiveEvents({1: 1, 2: 0}).get_dict()
{1: 1}
```

```
>>> weird = dt.WeightedDie({1: 1, 2: 0})
>>> weird.get_dict()
{1: 1}
>>> weird.get_size()
2
>>> weird.get_raw_dict() == {1: 1, 2: 0}
True
```

7.4.2 errors for dice

Top

Special rule for WeightedDie and ModWeightedDie

```
>>> dt.WeightedDie({0: 1})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: rolls may not be less than 1. use ModWeightedDie
```

```
>>> dt.ModWeightedDie({0: 1}, 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: rolls may not be less than 1. use ModWeightedDie
```

Here's how to add 0 one time (which does nothing, btw)

```
>>> dt.ModWeightedDie({1: 1}, -1).get_dict()
{0: 1}
```

StrongDie also has a weird case that can be unpredictable. Basically, don't multiply by zero

```
>>> table = dt.DiceTable.new().add_die(dt.Die(6))
```

```
>>> table = table.add_die(dt.StrongDie(dt.Die(100), 0), 100)
```

```
>>> table.get_dict() == {1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1}
True
```

```
>>> print(table)
1D6
(100D100)X(0)
```

```
>>> stupid_die = dt.StrongDie(dt.ModWeightedDie({1: 2, 3: 4}, -1), 0)
>>> table = table.add_die(stupid_die, 2)  # this rolls zero with weight 4
>>> print(table)
(2D3-2  W:6)X(0)
1D6
(100D100)X(0)
>>> table.get_dict() ==  {1: 16, 2: 16, 3: 16, 4: 16, 5: 16, 6: 16}  # this is␣
↪correct, it's just stupid.
True
```

ExplodingOn will raise an error if the values in "explodes_on" are not in input_die.get_dict()

```
>>> input_die = dt.WeightedDie({1: 2, 3: 1, 5: 1, 7: 2})
>>> dt.ExplodingOn(input_die, ()).get_dict() == {1: 72, 3: 36, 5: 36, 7: 72}
True
>>> dt.ExplodingOn(input_die, (2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: "explodes_on" value not present in input_die.get_dict()
```

7.4.3 add_die and remove_die are relatively safe

Top

dicetables.dicetable.DiceTable.add_die() and *dicetables.dicetable.DiceTable.add_die()* are safe. They raise an error if you remove too many dice or add or remove a negative number.

If you "remove" or "combine" with a negative number, nothing should happen, but i make no guarantees.

If you use "remove" to remove what you haven't added, it may or may not raise an error, but it's guaranteed buggy.

Here are "add_die" and "remove_die" failing fast:

```
>>> table = dt.DiceTable.new().add_die(dt.Die(6))
```

```
>>> table = table.remove_die(dt.Die(6), 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
DiceRecordError: Tried to create a DiceRecord with a negative value at Die(6): -3
```

```
>>> table = table.remove_die(dt.Die(10))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
DiceRecordError: Tried to create a DiceRecord with a negative value at Die(10): -1
```

```
>>> table = table.add_die(dt.Die(6), -3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
DiceRecordError: Tried to add_die or remove_die with a negative number.
```

```
>>> table = table.remove_die(dt.Die(6), -3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
DiceRecordError: Tried to add_die or remove_die with a negative number.
```

7.4.4 combine and remove are not

Top

And now, this is the trouble you can get into with *dicetables.additiveevents.AdditiveEvents.combine()* and *dicetables.additiveevents.AdditiveEvents.remove()*

```
>>> table = dt.DiceTable.new().add_die(dt.Die(6))
>>> table.get_dict() == {1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1}
True
>>> table = table.combine(dt.Die(10000), -100)
>>> table.get_dict() == {1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1}
True
>>> table = table.remove(dt.Die(2), 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: min() arg is an empty sequence <-didn't know this would happen, but at␣
↪least failed loudly
```

```
>>> table = table.remove(dt.Die(2), 2)
```

```
>>> table.get_dict() == {-1: 1, 1: 1}  # bad. this is a random answer
True
```

(I know why you're about to get wacky and inaccurate errors, and I could fix the bug, except . . . YOU SHOULD NEVER EVER DO THIS!!!!)

```
>>> table = table.remove(dt.AdditiveEvents({-5: 100}))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
EventsFactoryError: Error Code: SIGNATURES DIFFERENT
Factory:    <class 'dicetables.factory.eventsfactory.EventsFactory'>
Error At:   <class 'dicetables.dicetable.DiceTable'>
Attempted to construct a class already present in factory, but with a different␣
↪signature.
Class: <class 'dicetables.dicetable.DiceTable'>
Signature In Factory: ('get_dict', 'dice_data')
To reset the factory to its base state, use EventsFactory.reset()
```

Calling combine_by_flattened_list can be risky

```
>>> x = dt.AdditiveEvents({1:1, 2: 5})
>>> x = x.combine_by_flattened_list(dt.AdditiveEvents({1: 2, 3: 4}), 5)
>>> x = x.combine_by_flattened_list(dt.AdditiveEvents({1: 2, 3: 4*10**10}), 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

```
>>> x = x.combine_by_flattened_list(dt.AdditiveEvents({1: 2, 3: 4*10**700}))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
OverflowError: cannot fit 'int' into an index-sized integer
```

7.4.5 making a DiceTable with nonsense

Top

Since you can instantiate a DiceTable with any legal input, you can make a table with utter nonsense. It will work horribly. for instance, the dictionary for 2D6 is:

{2: 1, 3: 2, 4: 3, 5: 4, 6: 5, 7: 6, 8: 5, 9: 4, 10: 3, 11: 2, 12: 1}

```
>>> nonsense = dt.DiceTable({1: 1}, dt.DiceRecord({dt.Die(6): 2}))  # <- BAD DATA!!!!
>>> print(nonsense)  # <- the dice record says it has 2D6, but the events dictionary␣
↪is WRONG
2D6
>>> nonsense = nonsense.remove_die(dt.Die(6), 2)  # <- so here's your error. I hope␣
↪you're happy.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: min() arg is an empty sequence
```

But, you cannot instantiate a DiceTable with negative values for dice. And you cannot instantiate a DiceTable with non-sense values for dice.

```
>>> dt.DiceTable({1: 1}, dt.DiceRecord({dt.Die(3): 3, dt.Die(5): -1}))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
DiceRecordError: Tried to create a DiceRecord with a negative value at Die(5): -1
```

```
>>> dt.DiceTable({1: 1}, dt.DiceRecord({'a': 2.0}))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
DiceRecordError: input must be {ProtoDie: int, ...}
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

F

G

I

L