

---

# **sudokutools Documentation**

***Release 0.3.1***

**Maik Messerschmidt**

**Jun 21, 2018**



---

## Contents

---

<b>1</b>	<b>README</b>	<b>3</b>
<b>2</b>	<b>Contents</b>	<b>5</b>
2.1	The sudokutools library . . . . .	5
2.2	License . . . . .	19
<b>3</b>	<b>Indices and tables</b>	<b>21</b>
	<b>Python Module Index</b>	<b>23</b>



Yet another python sudoku library.



# CHAPTER 1

---

## README

---

You can find a short introduction on the [project site](#).





## 2.1 The sudokutools library

Yet another python sudoku library.

sudokutools is a collection of functions and classes, which enable you to read, create, analyze, solve and print sudokus.

### Package modules:

- sudokutools.generate: Create new sudokus.
- sudokutools.solve: Low-level solving and checking of sudokus.
- sudokutools.sudoku: Parse, print and compare sudokus.

### 2.1.1 sudokutools.sudoku - Parsing, printing and comparing

Parse, print and compare sudokus.

#### Classes defined here:

- Sudoku: Represents a sudoku.

#### Functions defined here:

- view(): Return sudoku (with candidates) as a human-readable string.

**class** sudokutools.sudoku.Sudoku (size=(3, 3))

Bases: object

Represents a sudoku.

This class provides methods to read, write, copy and display data to and from a sudoku as well as compare one sudoku with another. It stores 9x9 fields and the numbers and candidates within these fields.

Coordinates for row and column access are values from 0 to 8 (including). Using other values will most likely raise an IndexError (even though negative numbers are supported, they should not be used).

Overview:

**Data access (read, write):**

- `__getitem__()`
- `__setitem__()`
- `get_candidates()`
- `set_candidates()`
- `remove_candidates()`

**Copying:**

- `copy()`

**Comparing:**

- `empty()`
- `__len__()`
- `__eq__()`
- `diff()`

**Printing:**

- `__str__()`
- `encode()`

**Parsing:**

- `decode()`

`__eq__` (*other*)

Return if other is equal in all fields.

**Parameters** *other* (*Sudoku*) – Most likely another *Sudoku* instance, but any object, which can be accessed by `other[row, col]` is valid.

**Returns**

**True, if all fields are equal and false if not or *other* is an incompatible type.**

**Return type** `bool`

`__getitem__` (*key*)

Return the number in the field referenced by key.

**Parameters** *key* (*int*, *int*) – row and column of the requested field. Must be in range(0, width \* height).

**Returns** The number in the given field, 0 representing an empty field.

**Return type** `int`

**Raises** `IndexError` – if the given coordinates are not valid.

`__iter__` ()

Iterate through all coordinates of the sudoku.

Yields: (*int*, *int*): row and column of each field.

`__len__` ()

Return the number of fields in this sudoku.

**Returns** The number of fields in this sudoku.

**Return type** int

**\_\_getitem\_\_** (*key, value*)

Set the number in the field referenced by key to value.

**Parameters**

- **key** (*int, int*) – row and column of the requested field. Must be in range(0, width \* height).
- **value** (*int*) – The number to set the field to, 0 representing an empty field.

**Raises** `IndexError` – if the given coordinates are not valid.

**\_\_str\_\_** ()

Return sudoku as a human-readable string.

**Returns** String representing the sudoku.

**Return type** str

**\_quad\_without\_row\_and\_column\_of** (*row, col*)

Return some coordinates in the square of (col, row) as a list.

The coordinates in the same row and column are removed.

This is an internal function and should not be used outside of the sudoku module.

**box\_at** (*row, col*)

Return the box index of the field at (row, col)

**Parameters**

- **row** (*int*) – The row of the field.
- **col** (*int*) – The column of the field.

**Returns** The index of the box, in which the field (row, col) lies.

**Return type** int

**box\_of** (*row, col, include=True*)

Return all coordinates in the region of (col, row) as a list.

**Parameters**

- **row** (*int*) – The row of the field.
- **col** (*int*) – The column of the field.
- **include** (*bool*) – Whether or not to include (row, col).

**Returns**

**list of pairs (row, column) of all fields in** the same box (region).

**Return type** list of (int, int)

**column\_of** (*row, col, include=True*)

Return all coordinates in the column of (col, row) as a list.

**Parameters**

- **row** (*int*) – The row of the field.
- **col** (*int*) – The column of the field.

- **include** (*bool*) – Whether or not to include (row, col).

## Returns

**list of pairs (row, column) of all fields in** the same column.

**Return type** list of (int, int)

**copy** (*include\_candidates=False*)

Returns a copy of this sudoku.

**Parameters** `include_candidates` (*bool*) – Whether to copy candidates as well.

**Returns** The new sudoku instance.

**Return type** *Sudoku*

```
count ( )
```

Return the number of filled fields.

**Returns** number of filled fields.

**Return type** int

```
classmethod decode(s, empty='0', number_sep=None, sudoku_sep='|', candidate_sep=', ',
                    size=None)
```

Create a new sudoku from the string s.

## Parameters

- **s** (*str*) – A string representing the sudoku (see below).
- **empty** (*char*) – A character representing empty fields.
- **sudoku\_sep** (*char*) – A character, which separates field information from candidate information.
- **candidate\_sep** (*char*) – A character separating the candidate lists.
- **size** (*int*, *int*) – (width, height) of the new sudoku. If not given This will be calculated automatically, which may lead to wrong results. (It will always work as intended if width == height.)

**Returns** The newly created sudoku.

**Return type** *Sudoku*

**Examples for s:** 000030000005009602008004013020060000703040106000080090210300800306800700000020000  
000030000005009602008004013020060000703040106000080090210300800306800700000020000|124.235

Whitespace is ignored while parsing the string, so you can place newlines for better readability.

Each number represents the value of a column. If a row is full, we continue in the next one. So the sudoku above looks like this:

			3			
5			9		6	2
8			4		1	3
2			6			
7	3		4		1	6
			8		9	
2	1		3		8	

(continues on next page)

(continued from previous page)

3	6		8			7
			2			

The second string additionally defines candidates. Each set of candidates is separated by ‘,’ so the string above defines the candidates for (0, 0) to be 1, 2 and 4 and for (0, 1) to be 2, 3 and 5

This is the default format, which `encode()` uses and no other format is supported right now.

#### **diff** (*other*)

Iterate through coordinates with different values in *other*.

Compares each field in *other* to the corresponding field in *self* and yields the coordinates, if the number within is different.

**Parameters** *other* (*Sudoku*) – Most likely another *Sudoku* instance, but any object, which can be accessed by *other*[*row*, *col*] is valid.

**Yields** (*int*, *int*) – row and column of the next different field.

#### **empty** ()

Iterate through the coordinates of all empty fields.

**Yields** (*int*, *int*) – row and column of the next empty field.

#### **encode** (*row\_sep*=“, *col\_sep*=“, *include\_candidates*=*False*)

Return sudoku as a (machine-readable) string.

This method is mainly provided to output sudokus in a machine-readable format, but can be used for creating nicely looking representations as well.

#### **Parameters**

- **row\_sep** (*str*) – Separator between rows.
- **col\_sep** (*str*) – Separator between columns.
- **include\_candidates** –

**Returns** String representing this sudoku.

**Return type** (*str*)

For examples of default output string see `decode()`.

#### **filled** ()

Iterate through the coordinates of filled fields.

**Yields** (*int*, *int*) – row and column of the next filled field.

#### **get\_candidates** (*row*, *col*)

Return the candidates of the field at (*row*, *col*).

#### **Parameters**

- **row** (*int*) – The row of the field.
- **col** (*int*) – The column of the field.

**Returns** The candidates at (*row*, *col*).

**Return type** *frozenset*

#### **get\_number** (*row*, *col*)

Same as *sudoku*[*row*, *col*].

**remove\_candidates** (*row*, *col*, *value*)

Remove the given candidates in the field at (row, col).

Ignores candidates, which are not present in the field.

**Parameters**

- **row** (*int*) – The row of the field.
- **col** (*int*) – The column of the field.
- **value** (*iterable*) – The candidates to remove.

**row\_of** (*row*, *col*, *include=True*)

Return all coordinates in the row of (col, row) as a list.

**Parameters**

- **row** (*int*) – The row of the field.
- **col** (*int*) – The column of the field.
- **include** (*bool*) – Whether or not to include (row, col).

**Returns**

**list of pairs (row, column) of all fields in** the same row.

**Return type** list of (int, int)

**set\_candidates** (*row*, *col*, *value*)

Set the candidates of the field at (row, col) to value.

**Parameters**

- **row** (*int*) – The row of the field.
- **col** (*int*) – The column of the field.
- **value** (*iterable*) – The candidates to set the field to.

**set\_number** (*row*, *col*, *value*)

Same as `sudoku[row, col] = value`.

**surrounding\_of** (*row*, *col*, *include=True*)

Return all surrounding coordinates of (col, row) as a list.

**Parameters**

- **row** (*int*) – The row of the field.
- **col** (*int*) – The column of the field.
- **include** (*bool*) – Whether or not to include (row, col).

**Returns**

**list of pairs (row, column) of all fields in** the same column, row or square.

**Return type** list of (int, int)

`sudokutools.sudoku.view` (*sudoku*, *include\_candidates=True*, *number\_sep=None*, *candidate\_prefix='\*', align\_right=True*)

Return sudoku as a human-readable string.

**Parameters**

- **sudoku** – The sudoku to represent.
- **include\_candidates** (*bool*) – include candidates (or not)

- **number\_sep** (*str*) – separator for candidates. If set to None, this set to ‘,’, if there are numbers > 9 in the sudoku. Otherwise it will be the empty string.
- **candidate\_prefix** (*str*) – A string preceding the candidates. This is used to mark output as candidates (for example to recognize naked singles).
- **align\_right** (*bool*) – Align field content to the right (will be left-aligned, if set to False).

**Returns** String representing the sudoku.

**Return type** str

Example:

```
>>> from sudokutools.solve import init_candidates
>>> from sudokutools.sudoku import Sudoku, view
>>> sudoku = Sudoku.decode(''
... 003020600
... 900305001
... 001806400
... 008102900
... 700000008
... 006708200
... 002609500
... 800203009
... 005010300'')
>>> init_candidates(sudoku)
>>> print(view(sudoku))
*45  *4578  3 |  *49  2  *147 |  6  *5789  *57
  9  *24678  *47 |  3  *47  5 |  *78  *278  1
*25  *257  1 |  8  *79  6 |  4  *23579  *2357
-----+-----+-----
*345  *345  8 |  1  *3456  2 |  9  *34567  *34567
  7  *123459  *49 |  *459  *34569  *4 |  *1  *13456  8
*1345  *13459  6 |  7  *3459  8 |  2  *1345  *345
-----+-----+-----
*134  *1347  2 |  6  *478  9 |  5  *1478  *47
  8  *1467  *47 |  2  *457  3 |  *17  *1467  9
*46  *4679  5 |  *4  1  *47 |  3  *24678  *2467
```

## 2.1.2 sudokutools.solve - Solving and checking

Low-level solving and checking of sudokus.

**Functions defined here:**

- **bruteforce()**: Solves a sudoku using brute force.
- **calc\_candidates()**: Calculates candidates of a field in a sudoku.
- **init\_candidates()**: Sets the candidates for all fields in a sudoku.
- **find\_conflicts()**: Check sudoku for conflicting fields.
- **is\_unique()**: Check if a sudoku has exactly one solution.

`sudokutools.solve._do_bruteforce(sudoku)`

Solve sudoku `_inplace_` and yield it in a solved configuration.

This is an internal function and should not be used outside of the solve module.

`sudokutools.solve.bruteforce(sudoku)`

Solve the sudoku using brute force and yield solutions.

**Parameters** `sudoku` (`Sudoku`) – The Sudoku instance to solve.

**Yields** `Sudoku` – A solution of the sudoku.

`sudokutools.solve.calc_candidates(sudoku, row, col)`

Return a set of candidates of the sudoku at (row, col).

**Parameters**

- **sudoku** (`Sudoku`) – The Sudoku instance for which the candidates are calculated.
- **row** (`int`) – The row of the field
- **col** (`int`) – The column of the field.

**Returns** A set of candidates for the field at (row, col)

**Return type** `set`

`sudokutools.solve.find_conflicts(sudoku, *coords)`

Yield conflicts in sudoku at coords.

If coords is empty all possible coordinates will be searched.

**Parameters**

- **sudoku** (`Sudoku`) – The Sudoku instance to check.
- **coords** (*iterable of (int, int)*) – The coordinates to search within.

**Yields** *((int, int), (int, int), int)* –

**tuple of coordinate pairs and the** offending value.

E.g.: ((2, 3), (2, 6), 2) indicates, that there is a conflict for the fields (2, 3) and (2, 6) because both of them contain a 2.

`sudokutools.solve.init_candidates(sudoku, filled_only=False)`

Calculate and set all candidates in the sudoku.

Sets all candidates in the sudoku based on the numbers (and nothing else) in the surrounding fields.

**Parameters**

- **sudoku** (`Sudoku`) – The Sudoku instance for which the candidates are calculated.
- **filled\_only** (`bool`) – Only set candidate of already set fields. E.g. a field with a value of 2, will get the candidates {2}, but a field without a value will get no candidates.

`sudokutools.solve.is_unique(sudoku)`

Check if sudoku has exactly one solution.

**Parameters** `sudoku` (`Sudoku`) – The Sudoku instance to check.

**Returns** Whether or not the sudoku is unique.

**Return type** `bool`

### 2.1.3 sudokutools.solvers - High level solving

High level solving of sudokus.



This module provides classes which represent typical sudoku solving steps used by humans. Steps can be found and applied to a given sudoku. But steps can also be printed without applying them, e.g. to inform a user, what steps can be taken to solve the sudoku.

A single solve step may consist of multiple actions, e.g.

- Setting a number at a given field.
- Setting the candidates at a given field.
- Removing some of the candidates at a given field.

Solve steps defined here:

- CalcCandidates
- NakedSingle
- NakedPair
- NakedTriple
- NakedQuad
- NakedQuint
- HiddenSingle
- HiddenPair
- HiddenTriple
- HiddenQuad
- HiddenQuint
- PointingPair
- PointingTriple
- XWing
- Swordfish
- Jellyfish
- Bruteforce

**class** `sudokutools.solvers.Action`

Bases: `sudokutools.solvers.ActionTuple`

Named tuple, that represents a single change operation on a sudoku.

Create with: `Action(func, row, col, value)`

#### Parameters

- **func** (*callable*) – One of `Sudoku.set_number`, `Sudoku.set_candidates` and `Sudoku.remove_candidates`
- **row** (*int*) – The row of the field, which will be changed.
- **col** (*int*) – The column of the field, which will be changed.
- **value** (*int or iterable*) – The number or candidates to set/remove.

**class** `sudokutools.solvers.BasicFish` (*clues=()*, *affected=()*, *values=()*)

Bases: `sudokutools.solvers.SolveStep`

**build\_actions** (*sudoku*)

**classmethod find** (*sudoku*)

Iterates through all possible solve steps of this class.

**Parameters** *sudoku* (*Sudoku*) – The sudoku to solve.

**Yields** *SolveStep* – The next solve step.

**n** = 2

**class** *sudokutools.solvers.Bruteforce* (*row, col, value*)

Bases: *sudokutools.solvers.\_SingleFieldStep*

Solve the sudoku using brute force.

Bruteforce simply works by trial and error testing each combination of valid candidates in a field until a solution has been found.

**classmethod find** (*sudoku*)

Iterates through all possible solve steps of this class.

**Parameters** *sudoku* (*Sudoku*) – The sudoku to solve.

**Yields** *SolveStep* – The next solve step.

**class** *sudokutools.solvers.CalculateCandidates* (*clues=()*, *affected=()*, *values=()*)

Bases: *sudokutools.solvers.SolveStep*

Calculates the candidates of fields.

**build\_actions** (*sudoku*)

**classmethod find** (*sudoku*)

Iterates through all possible solve steps of this class.

**Parameters** *sudoku* (*Sudoku*) – The sudoku to solve.

**Yields** *SolveStep* – The next solve step.

**class** *sudokutools.solvers.HiddenPair* (*clues=()*, *affected=()*, *values=()*)

Bases: *sudokutools.solvers.HiddenTuple*

**n** = 2

**class** *sudokutools.solvers.HiddenQuad* (*clues=()*, *affected=()*, *values=()*)

Bases: *sudokutools.solvers.HiddenTuple*

**n** = 4

**class** *sudokutools.solvers.HiddenQuint* (*clues=()*, *affected=()*, *values=()*)

Bases: *sudokutools.solvers.HiddenTuple*

**n** = 5

**class** *sudokutools.solvers.HiddenSingle* (*row, col, value*)

Bases: *sudokutools.solvers.\_SingleFieldStep*

Finds hidden singles in a sudoku.

A hidden single is a field containing a candidate which exists in no other fields in the same row, column or box.

The field can be set to this candidate and this candidate can be removed from all fields in the same row, column and box.

**classmethod find** (*sudoku*)

Iterates through all possible solve steps of this class.

**Parameters** *sudoku* (*Sudoku*) – The sudoku to solve.

**Yields** *SolveStep* – The next solve step.

```
class sudokutools.solvers.HiddenTriple (clues=(), affected=(), values=())
    Bases: sudokutools.solvers.HiddenTuple
```

**n** = 3

```
class sudokutools.solvers.HiddenTuple (clues=(), affected=(), values=())
    Bases: sudokutools.solvers.SolveStep
```

Finds hidden tuples in a sudoku.

A hidden tuple is a set of n fields in a row, column or box, which (in unison) contain a set of at most n candidates, which are present in no other fields of the same row, column or box.

All other candidates can be removed from these fields.

```
build_actions (sudoku)
```

```
classmethod find (sudoku)
```

Iterates through all possible solve steps of this class.

**Parameters** **sudoku** (*Sudoku*) – The sudoku to solve.

**Yields** *SolveStep* – The next solve step.

**n** = 2

```
class sudokutools.solvers.Jellyfish (clues=(), affected=(), values=())
    Bases: sudokutools.solvers.BasicFish
```

**n** = 4

```
class sudokutools.solvers.NakedPair (clues=(), affected=(), values=())
    Bases: sudokutools.solvers.NakedTuple
```

**n** = 2

```
class sudokutools.solvers.NakedQuad (clues=(), affected=(), values=())
    Bases: sudokutools.solvers.NakedTuple
```

**n** = 4

```
class sudokutools.solvers.NakedQuint (clues=(), affected=(), values=())
    Bases: sudokutools.solvers.NakedTuple
```

**n** = 5

```
class sudokutools.solvers.NakedSingle (row, col, value)
    Bases: sudokutools.solvers._SingleFieldStep
```

Finds naked singles in a sudoku.

A naked single is a field with only one candidate.

The field can be set to this candidate and this candidate can be removed from all fields in the same row, column and box.

```
classmethod find (sudoku)
```

Iterates through all possible solve steps of this class.

**Parameters** **sudoku** (*Sudoku*) – The sudoku to solve.

**Yields** *SolveStep* – The next solve step.

```
class sudokutools.solvers.NakedTriple (clues=(), affected=(), values=())
    Bases: sudokutools.solvers.NakedTuple
```

**n = 3**

**class** sudokutools.solvers.**NakedTuple** (*clues=()*, *affected=()*, *values=()*)

Bases: *sudokutools.solvers.SolveStep*

Finds naked tuples in a sudoku.

A naked tuple is a set of n fields in a row, column or box, which (in unison) contain a set of at most n candidates.

These candidates can be removed from all fields in the same row, column or box.

**build\_actions** (*sudoku*)

**classmethod** **find** (*sudoku*)

Iterates through all possible solve steps of this class.

**Parameters** **sudoku** (*Sudoku*) – The sudoku to solve.

**Yields** *SolveStep* – The next solve step.

**n = 2**

**class** sudokutools.solvers.**PointingPair** (*clues=()*, *affected=()*, *values=()*)

Bases: *sudokutools.solvers.PointingTuple*

**n = 2**

**class** sudokutools.solvers.**PointingTriple** (*clues=()*, *affected=()*, *values=()*)

Bases: *sudokutools.solvers.PointingTuple*

**n = 3**

**class** sudokutools.solvers.**PointingTuple** (*clues=()*, *affected=()*, *values=()*)

Bases: *sudokutools.solvers.SolveStep*

**build\_actions** (*sudoku*)

**classmethod** **find** (*sudoku*)

Iterates through all possible solve steps of this class.

**Parameters** **sudoku** (*Sudoku*) – The sudoku to solve.

**Yields** *SolveStep* – The next solve step.

**n = 2**

**class** sudokutools.solvers.**SolveStep** (*clues=()*, *affected=()*, *values=()*)

Bases: object

**\_\_ge\_\_** (*other*)

$x.\_\text{ge}\_(y) \iff x \geq y$

**\_\_gt\_\_** (*other*)

$x.\_\text{gt}\_(y) \iff x > y$

**\_\_le\_\_** (*other*)

$x.\_\text{le}\_(y) \iff x \leq y$

**apply** (*sudoku*)

Apply this solve step to the sudoku.

**classmethod** **apply\_all** (*sudoku*)

Apply all possible steps of this class to the sudoku.

**build\_actions** (*sudoku*)

**classmethod find** (*sudoku*)

Iterates through all possible solve steps of this class.

**Parameters** *sudoku* (*Sudoku*) – The sudoku to solve.

**Yields** *SolveStep* – The next solve step.

**class** *sudokutools.solvers.Swordfish* (*clues=()*, *affected=()*, *values=()*)

Bases: *sudokutools.solvers.BasicFish*

**n** = 3

**class** *sudokutools.solvers.XWing* (*clues=()*, *affected=()*, *values=()*)

Bases: *sudokutools.solvers.BasicFish*

**n** = 2

**class** *sudokutools.solvers.\_SingleFieldStep* (*row, col, value*)

Bases: *sudokutools.solvers.SolveStep*

Represents a solve method, which sets a single field.

**build\_actions** (*sudoku*)

**classmethod find** (*sudoku*)

Iterates through all possible solve steps of this class.

**Parameters** *sudoku* (*Sudoku*) – The sudoku to solve.

**Yields** *SolveStep* – The next solve step.

*sudokutools.solvers.solve* (*sudoku, report=<function <lambda>>*)

Solve the sudoku and return the solution.

**Parameters**

- **sudoku** (*Sudoku*) – The sudoku to solve.
- **report** (*callable*) – A function taking a single argument (the current step), which can be used as a callback.

**Returns** The solution of the sudoku.

**Return type** *Sudoku*

## 2.1.4 *sudokutools.generate* - Creating new sudokus

Create new sudokus.

**Functions defined here:**

- *create\_solution*(): Create a complete sudoku without conflicts.
- *generate*(): Create a new sudoku.
- *generate\_from\_template*(): Create a new sudoku given a template pattern.

*sudokutools.generate.create\_solution* (*size=(3, 3)*)

Returns a sudoku, without empty or conflicting fields.

**Parameters** *size* (*int, int*) – region\_width and region\_height of the filled sudoku.

**Returns** The completely filled Sudoku instance.

**Return type** *Sudoku*

`sudokutools.generate.generate(min_count=0, symmetry=None, size=(3, 3))`

Generate a sudoku and return it.

#### Parameters

- **min\_count** (*int*) – Number of fields that must be filled at least. Any number above 81 will raise a `ValueError`, Any number below 17 makes no sense (but will not cause an error), since unique sudokus must have at least 17 filled fields.
- **symmetry** (*str*) – The kind of symmetry that will be created. Possible values are: `None` (no symmetry), `“rotate-90”`, `“rotate-180”`, `“mirror-x”`, `“mirror-y”` and `“mirror-xy”`.
- **size** (*int*, *int*) – region\_width and region\_height of the filled sudoku.

**Returns** The generated `Sudoku` instance.

**Return type** `Sudoku`

#### Raises

- `ValueError`, if symmetry is not a valid argument.
- `ValueError`, if min\_count is larger then len(sudoku).

`sudokutools.generate.generate_from_template(template, tries=100)`

Create a new sudoku from a given template.

#### Parameters

- **template** (`Sudoku`) – A sudoku, which describes the pattern to use. Every non-zero value of the template will be a filled field in the created solution.
- **tries** (*int*) – The number of tries until we give up. If `tries < 0`, the function will run, until a solution is found. Take note, that this may deadlock your program, if a solution is not possible.

**Returns** The created sudoku.

**Return type** `Sudoku`

**Raises** `RuntimeError` – if the sudoku couldn’t be created, within the given number of tries.

So symmetry isn’t enough for you and you want your sudokus to look like your favorite animal? Then this function is for you! `generate_from_template` takes the pattern from template and returns a valid sudoku, which matches this pattern (if possible).

#### Creating sudokus from templates is done in two steps:

1. Create a template (`Sudoku`) from the template string.
2. Hand over this template to this function.

Example for a template string:

```
111111111
100000001
100000001
100111001
100111001
100111001
100000001
100000001
100000001
111111111
```

Will create a sudoku like this:

1	2	6		9	4	8		3	7	5
7									4	
3									6	
-----+-----+-----										
9				8	1	2			3	
5				3	9	6			1	
2				4	5	7			8	
-----+-----+-----										
4									7	
8									2	
6	3	7		1	2	5		4	8	9

## 2.2 License

```

1 MIT License
2
3 Copyright (c) 2017-2018 Maik Messerschmidt
4
5 Permission is hereby granted, free of charge, to any person obtaining a copy
6 of this software and associated documentation files (the "Software"), to deal
7 in the Software without restriction, including without limitation the rights
8 to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
9 copies of the Software, and to permit persons to whom the Software is
10 furnished to do so, subject to the following conditions:
11
12 The above copyright notice and this permission notice shall be included in all
13 copies or substantial portions of the Software.
14
15 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
16 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
17 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
18 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
19 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
20 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
21 SOFTWARE.
```





## CHAPTER 3

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### S

- `sudokutools`, [5](#)
- `sudokutools.generate`, [17](#)
- `sudokutools.solve`, [11](#)
- `sudokutools.solvers`, [12](#)
- `sudokutools.sudoku`, [5](#)



## Symbols

\_SingleFieldStep (class in sudokutools.solvers), 17  
 \_\_eq\_\_() (sudokutools.sudoku.Sudoku method), 6  
 \_\_ge\_\_() (sudokutools.solvers.SolveStep method), 16  
 \_\_getitem\_\_() (sudokutools.sudoku.Sudoku method), 6  
 \_\_gt\_\_() (sudokutools.solvers.SolveStep method), 16  
 \_\_iter\_\_() (sudokutools.sudoku.Sudoku method), 6  
 \_\_le\_\_() (sudokutools.solvers.SolveStep method), 16  
 \_\_len\_\_() (sudokutools.sudoku.Sudoku method), 6  
 \_\_setitem\_\_() (sudokutools.sudoku.Sudoku method), 7  
 \_\_str\_\_() (sudokutools.sudoku.Sudoku method), 7  
 \_do\_bruteforce() (in module sudokutools.solve), 11  
 \_quad\_without\_row\_and\_column\_of() (sudokutools.sudoku.Sudoku method), 7

## A

Action (class in sudokutools.solvers), 13  
 apply() (sudokutools.solvers.SolveStep method), 16  
 apply\_all() (sudokutools.solvers.SolveStep class method), 16

## B

BasicFish (class in sudokutools.solvers), 13  
 box\_at() (sudokutools.sudoku.Sudoku method), 7  
 box\_of() (sudokutools.sudoku.Sudoku method), 7  
 Bruteforce (class in sudokutools.solvers), 14  
 bruteforce() (in module sudokutools.solve), 11  
 build\_actions() (sudokutools.solvers.\_SingleFieldStep method), 17  
 build\_actions() (sudokutools.solvers.BasicFish method), 13  
 build\_actions() (sudokutools.solvers.CalculateCandidates method), 14  
 build\_actions() (sudokutools.solvers.HiddenTuple method), 15  
 build\_actions() (sudokutools.solvers.NakedTuple method), 16  
 build\_actions() (sudokutools.solvers.PointingTuple method), 16

build\_actions() (sudokutools.solvers.SolveStep method), 16

## C

calc\_candidates() (in module sudokutools.solve), 12  
 CalculateCandidates (class in sudokutools.solvers), 14  
 column\_of() (sudokutools.sudoku.Sudoku method), 7  
 copy() (sudokutools.sudoku.Sudoku method), 8  
 count() (sudokutools.sudoku.Sudoku method), 8  
 create\_solution() (in module sudokutools.generate), 17

## D

decode() (sudokutools.sudoku.Sudoku class method), 8  
 diff() (sudokutools.sudoku.Sudoku method), 9

## E

empty() (sudokutools.sudoku.Sudoku method), 9  
 encode() (sudokutools.sudoku.Sudoku method), 9

## F

filled() (sudokutools.sudoku.Sudoku method), 9  
 find() (sudokutools.solvers.\_SingleFieldStep class method), 17  
 find() (sudokutools.solvers.BasicFish class method), 13  
 find() (sudokutools.solvers.Bruteforce class method), 14  
 find() (sudokutools.solvers.CalculateCandidates class method), 14  
 find() (sudokutools.solvers.HiddenSingle class method), 14  
 find() (sudokutools.solvers.HiddenTuple class method), 15  
 find() (sudokutools.solvers.NakedSingle class method), 15  
 find() (sudokutools.solvers.NakedTuple class method), 16  
 find() (sudokutools.solvers.PointingTuple class method), 16  
 find() (sudokutools.solvers.SolveStep class method), 16  
 find\_conflicts() (in module sudokutools.solve), 12

## G

generate() (in module sudokutools.generate), 17  
 generate\_from\_template() (in module sudokutools.generate), 18  
 get\_candidates() (sudokutools.sudoku.Sudoku method), 9  
 get\_number() (sudokutools.sudoku.Sudoku method), 9

## H

HiddenPair (class in sudokutools.solvers), 14  
 HiddenQuad (class in sudokutools.solvers), 14  
 HiddenQuint (class in sudokutools.solvers), 14  
 HiddenSingle (class in sudokutools.solvers), 14  
 HiddenTriple (class in sudokutools.solvers), 15  
 HiddenTuple (class in sudokutools.solvers), 15

## I

init\_candidates() (in module sudokutools.solve), 12  
 is\_unique() (in module sudokutools.solve), 12

## J

Jellyfish (class in sudokutools.solvers), 15

## N

n (sudokutools.solvers.BasicFish attribute), 14  
 n (sudokutools.solvers.HiddenPair attribute), 14  
 n (sudokutools.solvers.HiddenQuad attribute), 14  
 n (sudokutools.solvers.HiddenQuint attribute), 14  
 n (sudokutools.solvers.HiddenTriple attribute), 15  
 n (sudokutools.solvers.HiddenTuple attribute), 15  
 n (sudokutools.solvers.Jellyfish attribute), 15  
 n (sudokutools.solvers.NakedPair attribute), 15  
 n (sudokutools.solvers.NakedQuad attribute), 15  
 n (sudokutools.solvers.NakedQuint attribute), 15  
 n (sudokutools.solvers.NakedTriple attribute), 15  
 n (sudokutools.solvers.NakedTuple attribute), 16  
 n (sudokutools.solvers.PointingPair attribute), 16  
 n (sudokutools.solvers.PointingTriple attribute), 16  
 n (sudokutools.solvers.PointingTuple attribute), 16  
 n (sudokutools.solvers.Swordfish attribute), 17  
 n (sudokutools.solvers.XWing attribute), 17  
 NakedPair (class in sudokutools.solvers), 15  
 NakedQuad (class in sudokutools.solvers), 15  
 NakedQuint (class in sudokutools.solvers), 15  
 NakedSingle (class in sudokutools.solvers), 15  
 NakedTriple (class in sudokutools.solvers), 15  
 NakedTuple (class in sudokutools.solvers), 16

## P

PointingPair (class in sudokutools.solvers), 16  
 PointingTriple (class in sudokutools.solvers), 16  
 PointingTuple (class in sudokutools.solvers), 16

## R

remove\_candidates() (sudokutools.sudoku.Sudoku method), 9  
 row\_of() (sudokutools.sudoku.Sudoku method), 10

## S

set\_candidates() (sudokutools.sudoku.Sudoku method), 10  
 set\_number() (sudokutools.sudoku.Sudoku method), 10  
 solve() (in module sudokutools.solvers), 17  
 SolveStep (class in sudokutools.solvers), 16  
 Sudoku (class in sudokutools.sudoku), 5  
 sudokutools (module), 5  
 sudokutools.generate (module), 17  
 sudokutools.solve (module), 11  
 sudokutools.solvers (module), 12  
 sudokutools.sudoku (module), 5  
 surrounding\_of() (sudokutools.sudoku.Sudoku method), 10  
 Swordfish (class in sudokutools.solvers), 17

## V

view() (in module sudokutools.sudoku), 10

## X

XWing (class in sudokutools.solvers), 17