

---

**COWS**

***Release v1.0.0***

**Jun 07, 2019**



---

## Contents

---

<b>1</b>	<b>Data Structure Reference</b>	<b>1</b>
1.1	Dictionary . . . . .	1
1.2	List . . . . .	3
1.3	Set . . . . .	4
1.4	Trie . . . . .	5
<b>2</b>	<b>Motivation</b>	<b>9</b>
<b>3</b>	<b>Provided Data Structures</b>	<b>11</b>
3.1	<code>cows.List</code> . . . . .	11
3.2	<code>cows.Set</code> . . . . .	11
3.3	<code>cows.Dict</code> . . . . .	12
3.4	<code>cows.Trie</code> . . . . .	12
<b>4</b>	<b>Performance</b>	<b>15</b>
	<b>Python Module Index</b>	<b>17</b>
	<b>Index</b>	<b>19</b>



## 1.1 Dictionary

**class** cows.dictionary.Dict (*selector=None, updater=None, \*\*kwargs*)

Creates a dict-like object which checks has potentially ambiguous keys

This class provides a key/value store where the keys are strings and may contain wildcards. Unlike the builtin dict type where setting a key overwrites the existing associated value if it exists, this class allows for a user-defined updating function, *updater*. Since a given key may match more than one key in the Dict (due to wildcards), a *selector* function will be passed the list of matches which will select which to pass to *updater*.

### Parameters

- **selector** (*func*) – Called when `__setitem__` is called with *key* and a (possibly wildcard) match to *key* exists.

Must accept one argument, an iterable of (*key*, *value*) matches, and return a single element from the iterator that will be updated with *updater*.

- **updater** (*func*) – Called when `__setitem__` is called with *key* and *value* and a (possibly ambiguous) match to *key* exists.

Must accept three arguments *match*, *current\_value*, and *new\_value*. *match* and *current\_value* will be passed the key and value returned by the *selector* and *new\_value* will be passed the value passed to `__setitem__`.

Returns the value to set the value associated with *match* to.

- **\*\*kwargs** – Passed to underlying Trie

### Example

```
import cows

def increment(match, old_value, new_value):
```

(continues on next page)

(continued from previous page)

```
    return old_value + new_value

my_dict = cows.Dict(updater=increment)
my_dict['ABC'] = 1
my_dict['DEF'] = 2
my_dict['AB*'] = 10

for k, v in sorted(my_dict.items()):
    print(f'{k} --> {v}')
```

This code would output:

```
ABC --> 11
DEF --> 2
```

Now consider a more complicated example:

```
...
my_dict = cows.Dict(updater=increment)
my_dict['ABC'] = 1
my_dict['*EF'] = 2
my_dict['GHF'] = 3
my_dict['G*F'] = 5
```

Here the setting of G\*F matches both \*EF and GHF. By default, the first lexicographic match (in this case \*EF) is chosen for update:

```
*EF --> 7
ABC --> 1
GHF --> 3
```

However, this behavior can be overridden by passing a function as the `selector` parameter. This function must take one parameter, `matches` which yields (key, value) pairs for each matching entry and return the key of the desired pair.

For example, this selector chooses the `_last_` match when sorted in lexicographic order:

```
...
def last_match(matches):
    return sorted(matches, key=lambda m: m[0], reverse=True)[0]

my_dict = cows.Dict(updater=increment, selector=last_match)
...
```

This will output:

```
*EF --> 2
ABC --> 1
GHF --> 8
```

`__getitem__(key)`  
Gets items matching key.

**Parameters** `key` (*str*) – The key string to match

**Yields** The values that match `key`. Order is not guaranteed.

**`__iter__()`**

Yields the keys in the dictionary

**`__len__()`**

Returns the number of elements in the dictionary.

**`__setitem__(key, value)`**

Sets a value in the dictionary.

Sets *key* to *value* if no match for *key* already exists. If matches do exist, one is selected with `self.selector` function and is optionally updated with the `self.updater` function.

#### Parameters

- **key** (*str*) – The key to set
- **value** (*obj*) – The value to set

**`items()`**

**Returns** (*key*, *value*) tuples for each association in the dictionary.

**`keys()`**

**Returns** The keys in the dictionary.

**`values()`**

**Returns** The values in the dictionary.

## 1.2 List

**`class cows.list.List`** (*iterable=None*)

A list for storing potentially ambiguous strings.

This class allows strings with ambiguous characters to be searched. Insertion via `append()`, `extend()`, and `insert()` function normally, simply inserting values into a list. Accessor methods `index()`, `count()`, and `__contains__()` all take into account ambiguous characters, however.

Example:

```
import cows

l = cows.List(['ABCD', 'ABC*', 'DEFG'])
print(l)
# prints: cows.List(['ABCD', 'ABC*', 'DEFG'])

l.insert(2, '****')

print(l)
# print: cows.List(['ABCD', 'ABC*', '****', 'DEFG'])

print(l.index('D***'))
# prints: 2

print(l.count('A***')) # 3
# prints: 3
```

**`__contains__(key)`**

Returns if *key* is in the list taking into account ambiguity

**`__iter__()`**  
Yields items in the list

**`__len__()`**  
Returns the number of elements in the list

**`append(value)`**  
Appends `value` to the list

**`count(value)`**  
Counts the number of times `value` occurs in the list.  
  
This method takes into account ambiguity.

**`extend(iterable)`**  
Appends all elements in `iterable` to the list

**`index(value, start=None, end=None)`**  
Finds the first index of `value` in the list.  
  
Determines if `value` is in the list taking into account ambiguity and returns the first matching index.  
  
If `start` and/or `end` is specified, only searches that portion of the list using the slice operator. If `value` is not found raises a `ValueError`.

### Example

```
l = cows.List(['ABCD', 'ABC*', '****', 'DEFG'])  
  
print(l.index('D***'))
```

The output of the print statement is 2 since the first match for `D***` is at position 2 (with a value of `****`).

#### Parameters

- **`value`** (*str*) – The value for which to search.
- **`start`** (*int*) – The minimum index to start searching.
- **`end`** (*int*) – The maximum index to search through

**Returns** The minimum index that matches `value`

**Raises** `ValueError` – If no matches for `value` are found.

**`insert(i, value)`**  
Inserts `value` at position `i` in the list

## 1.3 Set

**`class cows.set.Set(iterable=None, **kwargs)`**  
Creates a set-like object which checks for ambiguous inclusion.

This class provides a basic implementation of the `set`, a group of distinct (unique) values. Uniqueness is checked based on ambiguous strings so `ABC*` and `*BCD` would be considered equivalent.

#### Parameters

- **`iterable`** (*iterable*) – An optional set of elements with which to populate
- **`set.`** (*the*) –



- **\*\*kwargs** – Passed to underlying Trie

### Example

```
import cows

s = cows.Set()
s.add('ABCD')
s.add('*EFG')
s.add('T')
s.add('ABC*') # Matches ABCD, so not added
s.add('HEF*') # Matches *EFG, so not added

print(s)
```

Produces:

```
cows.Set(['*EFG', 'ABCD', 'T'])
```

**\_\_iter\_\_()**

Yields the elements in the set

**\_\_len\_\_()**

Returns the number of elements in the set

**add(element)**

Adds an element to the set.

**Parameters element** (*str*) – The element to add.

## 1.4 Trie

**class cows.trie.Trie** (*key=None, value=<object object>, wildcard='\*', initialize=None*)

A trie which has accessors for ambiguous lookups.

This class is the basis of all other cows classes. It stores *all* strings which have been inserted, not taking into account ambiguity. No special methods (starting & ending with double underscores) take into account ambiguity. To search the trie for ambiguous matches, use `get_matches()`.

### Example

```
import cows

t = cows.Trie()
t['ABCD'] = 1
t['DE*G'] = 5

print(f'Matches for ABC* {list(t.get_matches("ABC*"))}')
print(f'Matches for D*FG {list(t.get_matches("D*FG"))}')
```

Outputs:

```
Matches for ABC* [('ABCD', cows.Trie(D, 1))]
Matches for D*FG [('DE*G', cows.Trie(G, 5))]
```

**Parameters**

- **key** (*char*) – The character representing the trie node.
- **value** (*object*) – An arbitrary Python object representing the data at the trie node.
- **wildcard** (*char*) – The character representing ambiguity.
- **initialize** (*tuple*) – Pairs of values with which to initialize the trie.

---

**Note:** Consider using the other cows data structures, which are more intuitive, before using a Trie.

---

**\_\_getitem\_\_** (*key*)

Gets an item from the trie.

Searches the trie for *key*. Note this does **not** take into account ambiguity, and will only find an exact match. For ambiguous searching, use *get\_matches()*.

**Parameters** **key** (*str*) – The key to search for

**Returns** The matching *Trie* node if *key* was found, else *None*

**\_\_len\_\_** ()

Returns the number of nodes in the trie

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

Sets a key/value pair in the trie.

Sets the value of *key* to *value*. Note this will affect exactly one trie node and does not take into account ambiguity. For a data structure that implements setting with ambiguity use *Dict*.

**Parameters**

- **key** (*str*) – The key to set.
- **value** (*obj*) – The data to associate with *key*

**children\_matching** (*prefix*)

Gets all child nodes matching the single character prefix. If the character is a wildcard, it will return all children and if a wildcard is included in the children, it will be included.

For example, if the children are:

```
[Trie('A'), Trie('B'), Trie('C'), Trie('*')]
```

where *\** is the wildcard, passing *A* to this method will return:

```
[Trie('A'), Trie('*')].
```

**Parameters** **prefix** (*char*) – A single character for which to search within children.

**Yields** Child(ren) matching *prefix*

**Raises** *ValueError* – If *prefix* is not a string of exactly one character.

**get\_matches** (*key*)

Searches the trie for strings matching *key*.

**Example**

If the trie contains *ABCD*, *ABCA*, and *CBC\**, the key *ABC\** will return *ABCD* and *ABCA*.

**Parameters** **key** (*str*) – The string for which to search for matches in the trie

**Yields** (key, value) tuples for nodes that match key.

---

**Note:** The order of yielded matches is not defined and is not guaranteed to be consistent.

---

**items** (*extract\_values=False*)

Gets all items in the trie.

**Yields** (node\_key, node) pairs of all items.

**keys** ()

Yields the keys in the trie

**values** (*extract\_values=False*)

Yields the values in the trie

**cows** (collections for wildcard strings) is a Python library that provides efficient collection implementations where equality checking allows for wildcards in both the search string and the strings already in the collection.



## CHAPTER 2

---

### Motivation

---

`cows` was developed for a common problem in bioinformatics: given a set of DNA sequences with the alphabet A, T, C, G, along with a wildcard N (indicating that the base is unknown), find the unique sequences and perform some operation on them. Examples of the operation are: counting how many times each unique sequence occurs and generate a consensus sequence for each unique sequence.

For a simple example, for counting unique sequences consider the following input and desired output:

input	output
-----	-----
ATNG	ATNG 2 # Comprised of ATNG and ATCN
ATCN	ANNT 1
ANNT	GTTC 1
GTTC	

Notice this task requires comparing strings with wildcards not just in one string, but in both. For example, matching ATCN to ATNG requires that the third and fourth characters both be considered wildcards.

Naively one could pairwise compare the sequences, ignore the positions where either contains an N, and check if all other positions match. However, this quickly becomes intractable as it scales with the square of the number of sequences.

`cows` uses a modified implementation of `atrie` (`cows.trie`) to reduce this complexity to scale linearly with the number of sequences.



---

## Provided Data Structures

---

Below are examples for the data structures included with cows. Please see the documentation in [Data Structure Reference](#) for detailed API information.

### 3.1 cows.List

A `cows.list` is a simple list implementation where insertion functions similarly to the builtin `list` data structure, but accessor methods take into account ambiguity. For example:

```
l = cows.List(['ABCD', 'ABC*', '****', 'DEFG'])

print(l.index('D***'))
```

The print statement outputs 2 since the first match for `D***` is at position 2 (with a value of `****`).

### 3.2 cows.Set

A `cows.set` stores unique strings similar to the builtin `set` data structure. Instead of using hashes for equality checks, the underlying `cows.trie` is used to check if the pattern being inserted matches any existing member of the set, taking into account wildcards in both. For example:

```
import cows

s = cows.Set(wildcard='*')
s.add('ABCD')
s.add('*EFG')
s.add('T')
s.add('ABC*') # Matches ABCD, so not added
s.add('HEF*') # Matches *EFG, so not added

print(s)
```

Produces:

```
cows.Set(['*EFG', 'ABCD', 'T'])
```

### 3.3 cows.Dict

cows dictionaries are similar to the builtin `dict` type insofar as they are key/value stores. They have a few key differences, however

First, when setting a value, if there is an existing (potentially ambiguous) match already in the dictionary, you can set an `updater` function to update the existing value, rather than simply overwrite it. Further, when inserting a key, because of ambiguity, multiple existing keys may match. Providing a `selector` function lets you define to which of the matches the `updater` should be applied.

See `cows.dictionary` for more detailed information.

```
import cows

def increment(match, old_value, new_value):
    return old_value + new_value

my_dict = cows.Dict(updater=increment)
my_dict['ABC'] = 1
my_dict['DEF'] = 2
my_dict['AB*'] = 10

for k, v in sorted(my_dict.items()):
    print(f'{k} --> {v}')
```

Produces:

```
ABC --> 11
DEF --> 2
```

### 3.4 cows.Trie

---

**Note:** Generally the `cows.trie` data structure shouldn't be used directly. Consider using one of its abstractions.

---

All other cows data structures are based on the `cows.trie` class. It allows for ambiguous queries taking into account wildcards both in the query string and elements in the trie.

An example of it's use:

```
import cows

t = cows.Trie()
t['ABCD'] = 1
t['DE*G'] = 5

print(f'Matches for ABC* {list(t.get_matches("ABC*"))}')
print(f'Matches for D*FG {list(t.get_matches("D*FG"))}')
```



**Outputs:**

```
Matches for ABC* [('ABCD', cows.Trie(D, 1))]  
Matches for D*FG [('DE*G', cows.Trie(G, 5))]
```



## CHAPTER 4

---

### Performance

---

cows is performant, requiring  $O(n)$  time for insertions and lookups with an input size of  $n$  strings. The naive approach which is currently quite common involves pairwise comparing the sequences in a collection resulting in  $O(n^2)$ , quickly becoming intractable.



### C

- `cows.dictionary`, 1
- `cows.list`, 3
- `cows.set`, 4
- `cows.trie`, 5



## Symbols

`__contains__()` (*cows.list.List method*), 3  
`__getitem__()` (*cows.dictionary.Dict method*), 2  
`__getitem__()` (*cows.trie.Trie method*), 6  
`__iter__()` (*cows.dictionary.Dict method*), 2  
`__iter__()` (*cows.list.List method*), 3  
`__iter__()` (*cows.set.Set method*), 5  
`__len__()` (*cows.dictionary.Dict method*), 3  
`__len__()` (*cows.list.List method*), 4  
`__len__()` (*cows.set.Set method*), 5  
`__len__()` (*cows.trie.Trie method*), 6  
`__setitem__()` (*cows.dictionary.Dict method*), 3  
`__setitem__()` (*cows.trie.Trie method*), 6

## A

`add()` (*cows.set.Set method*), 5  
`append()` (*cows.list.List method*), 4

## C

`children_matching()` (*cows.trie.Trie method*), 6  
`count()` (*cows.list.List method*), 4  
`cows.dictionary` (*module*), 1  
`cows.list` (*module*), 3  
`cows.set` (*module*), 4  
`cows.trie` (*module*), 5

## D

`Dict` (*class in cows.dictionary*), 1

## E

`extend()` (*cows.list.List method*), 4

## G

`get_matches()` (*cows.trie.Trie method*), 6

## I

`index()` (*cows.list.List method*), 4  
`insert()` (*cows.list.List method*), 4  
`items()` (*cows.dictionary.Dict method*), 3

`items()` (*cows.trie.Trie method*), 7

## K

`keys()` (*cows.dictionary.Dict method*), 3  
`keys()` (*cows.trie.Trie method*), 7

## L

`List` (*class in cows.list*), 3

## S

`Set` (*class in cows.set*), 4

## T

`Trie` (*class in cows.trie*), 5

## V

`values()` (*cows.dictionary.Dict method*), 3  
`values()` (*cows.trie.Trie method*), 7