Practical Computing for Scientists

Armin Sobhani

CSCI 2000U

UOIT – Fall 2015

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Python

Sets and Dictionaries

Storage

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Let's try an experiment

Let's try an experiment

```
>>> things = set()
>>> things.add('a string')
>>> print(things)
set(['a string'])
```

Let's try an experiment

```
>>> things = set()
>>> things.add('a string')
>>> print(things)
set(['a string'])

>>> things.add([1, 2, 3])
TypeError: unhashable type: 'list'
```

Let's try an experiment

```
>>> things = set()
>>> things.add('a string')
>>> print things
set(['a string'])

>>> things.add([1, 2, 3])
TypeError: unhashable type: 'list'
```

What's wrong?

Let's try an experiment

```
>>> things = set()
>>> things.add('a string')
>>> print things
set(['a string'])

>>> things.add([1, 2, 3])
TypeError: unhashable type: 'list'
```

What's wrong?

And what does the error message mean?

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

How are sets stored in a computer's memory?

How are sets stored in a computer's memory?

Could use a list

How are sets stored in a computer's memory?

Could use a list

```
def set_create():
  return []
```

How are sets stored in a computer's memory?

Could use a list

```
def set_create():
  return []

def set_in(set_list, item):
  for thing in set_list:
    if thing == item:
      return True
  return False
```

```
def set_add(set_list, item):
  for thing in set_list:
    if thing == item:
      return
  set.append(item)
```

```
def set_add(set_list, item):
  for thing in set_list:
    if thing == item:
      return
  set.append(item)
```

How efficient is this?

```
def set_add(set_list, item):
  for thing in set_list:
    if thing == item:
      return
  set.append(item)
```

How efficient is this?

With N items in the set, in and add take 1 to N steps

```
def set_add(set_list, item):
  for thing in set_list:
    if thing == item:
      return
  set.append(item)
```

How efficient is this?

With N items in the set, in and add take 1 to N steps

"Average" is N/2

```
def set_add(set_list, item):
  for thing in set_list:
    if thing == item:
      return
  set.append(item)
```

How efficient is this?

With N items in the set, in and add take 1 to N steps

"Average" is N/2

It's possible to do much better

```
def set_add(set_list, item):
  for thing in set_list:
    if thing == item:
      return
  set.append(item)
```

How efficient is this?

With N items in the set, in and add take 1 to N steps

"Average" is N/2

It's possible to do much better

But the solution puts some constraints on programs

UNIVERSITY
OF ONTARIO
INSTITUTE OF TECHNOLOGY

Start simple: how do we store a set of integers?

Start simple: how do we store a set of integers?

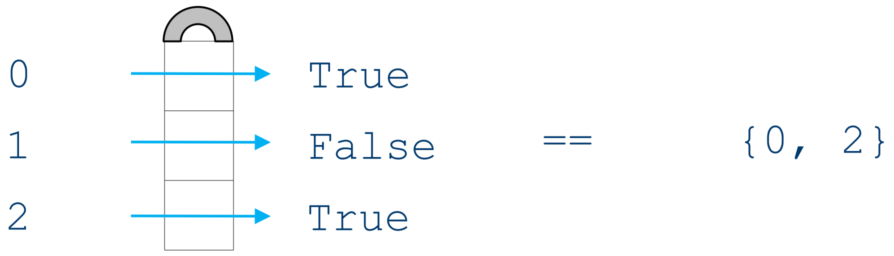
If the range of possible values is small and fixed,

use a list of Boolean flags ("present" or "absent")

Start simple: how do we store a set of integers?

If the range of possible values is small and fixed,

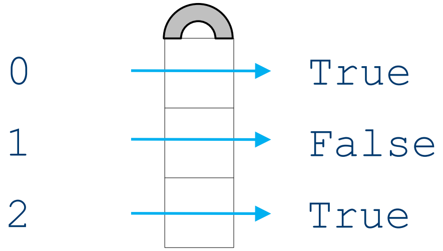
use a list of Boolean flags ("present" or "absent")



```
0    True
1    False    ==    {0, 2}
2    True
```

Start simple: how do we store a set of integers?

If the range of possible values is small and fixed, use a list of Boolean flags ("present" or "absent")



```
0  True
1  False     ==     {0, 2}
2  True
```

But what if the range of values is large, or can change over time?

Use a fixed-size *hash table* of length L

Use a fixed-size *hash table* of length L

Store the integer I at location I % L

Use a fixed-size *hash table* of length L

Store the integer I at location I % L

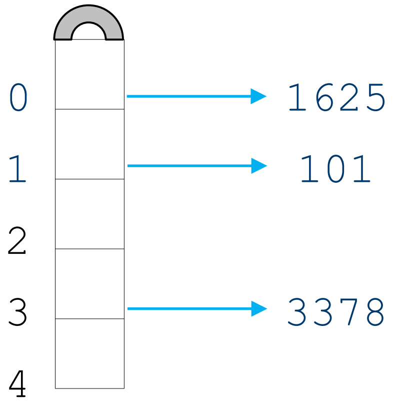
'%' is the remainder operator

Use a fixed-size *hash table* of length L

Store the integer I at location I % L

'%' is the remainder operator

{3378, 1625, 101} ==



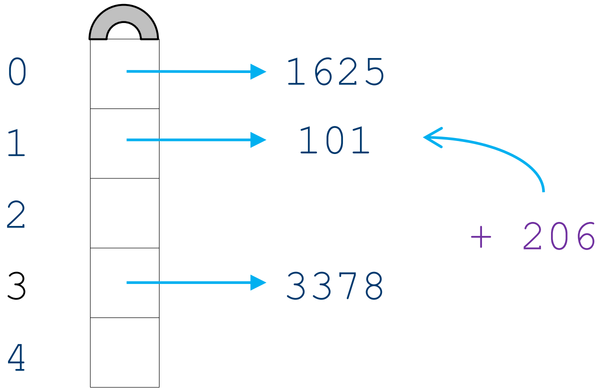
Time to insert or look up is constant (!)

Time to insert or look up is constant (!)

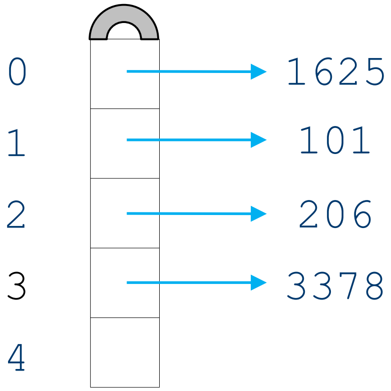
But what do we do when there's a collision?

Time to insert or look up is constant(!)

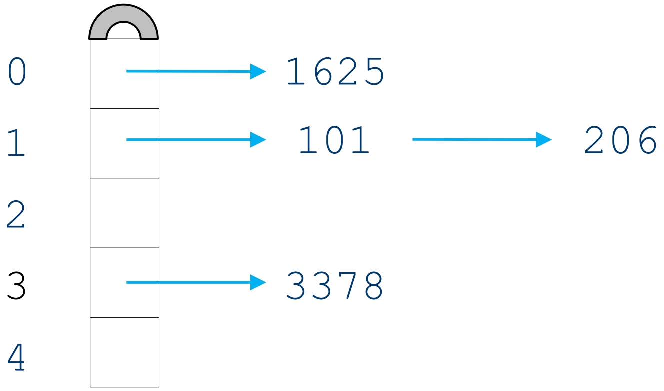
But what do we do when there's a collision?



Option #1: store it in the next empty slot



Option #2: chain values together



Either works well until the table is about 3/4 full

Either works well until the table is about 3/4 full

Then average time to look up/insert rises rapidly

Either works well until the table is about 3/4 full

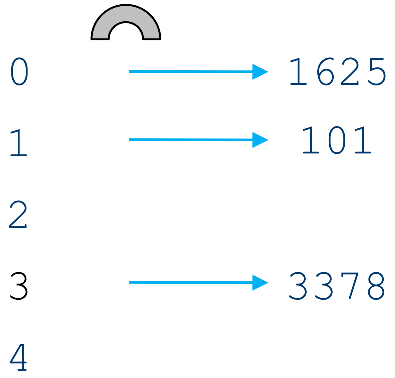
Then average time to look up/insert rises rapidly

So enlarge the table

Either works well until the table is about 3/4 full

Then average time to look up/insert rises rapidly

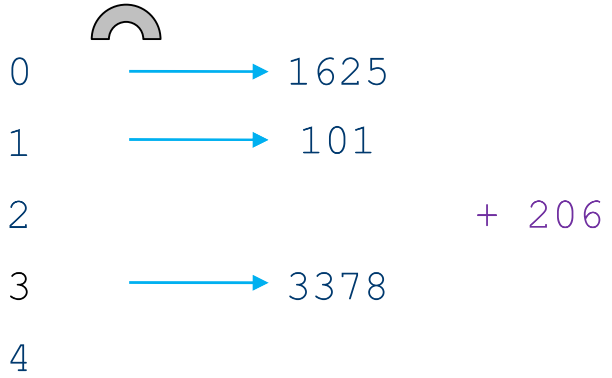
So enlarge the table



Either works well until the table is about 3/4 full

Then average time to look up/insert rises rapidly

So enlarge the table



0 → 1625

1 → 101

2 + 206

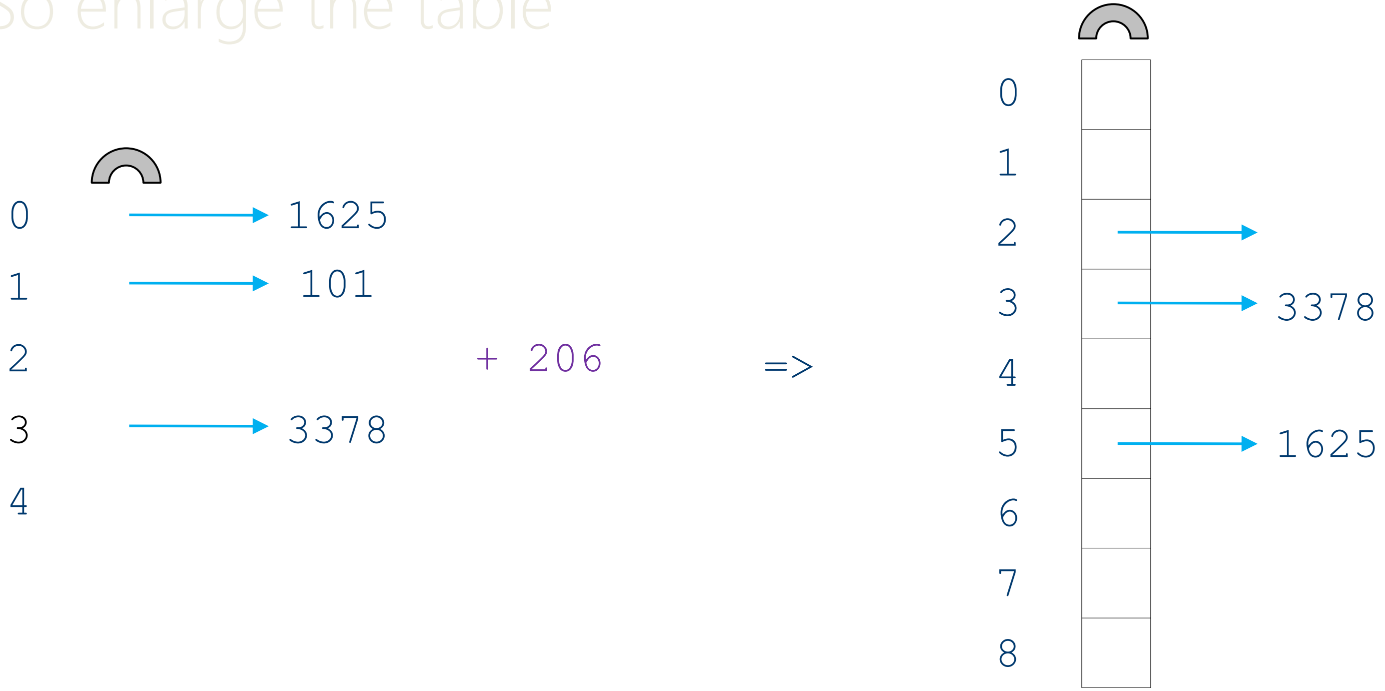
3 → 3378

4

Either works well until the table is about 3/4 full

Then average time to look up/insert rises rapidly

So enlarge the table



0 → 1625

1 → 101

2

3 → 3378

4

+ 206

=>

0

1

2 →

3 → 3378

4

5 → 1625

6

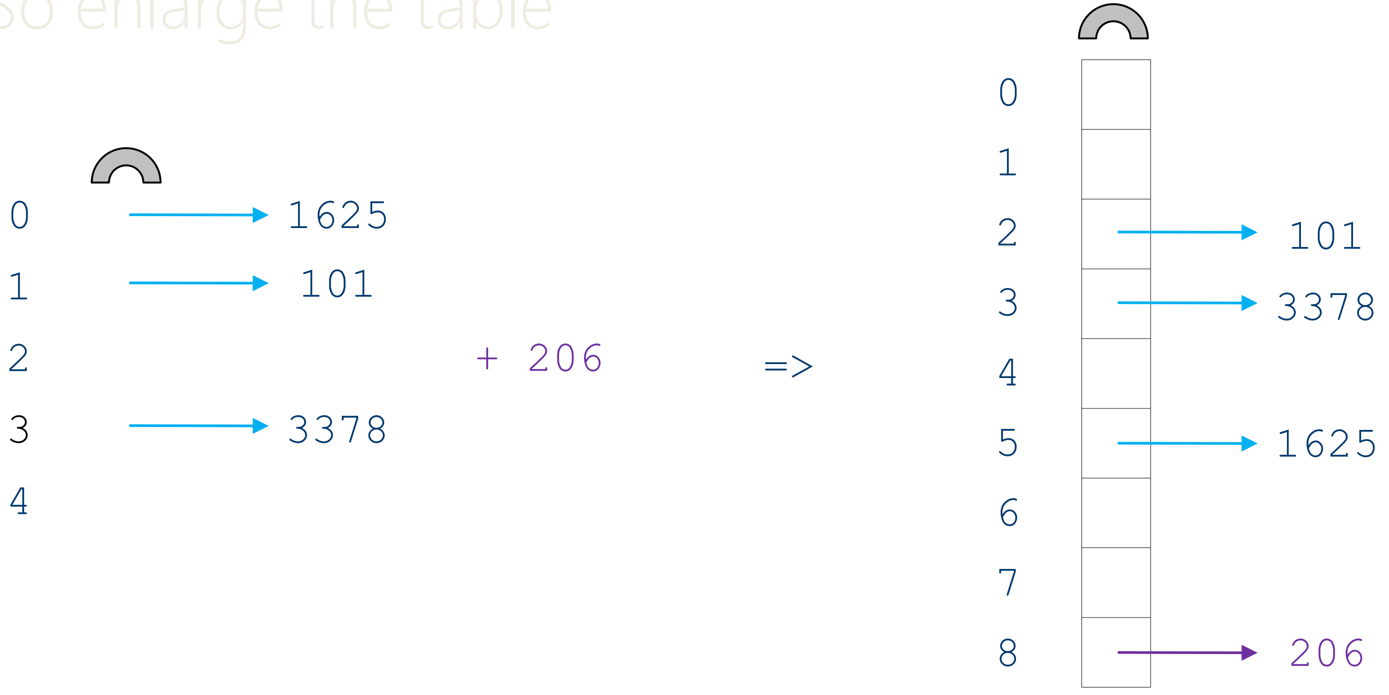
7

8

Either works well until the table is about 3/4 full

Then average time to look up/insert rises rapidly

So enlarge the table



How do we store strings?

How do we store strings?

Use a hash function to generate an integer index

based on the characters in the string

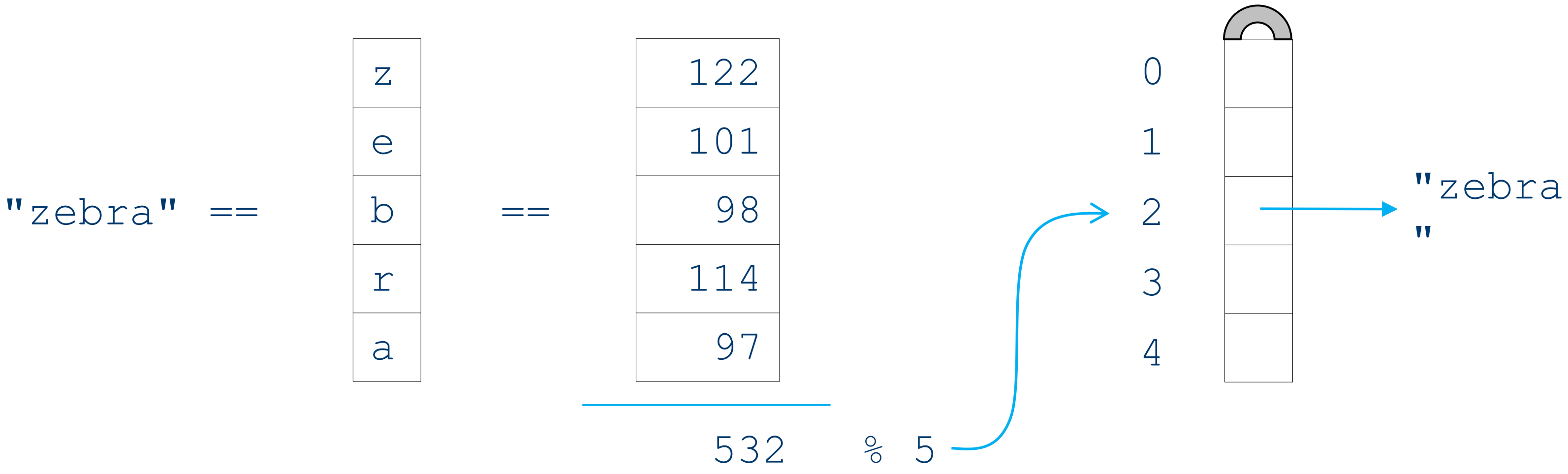
```
"zebra"
```

```
"zebra" ==
```

z
e
b
r
a

	z		122
	e		101
"zebra" ==	b	==	98
	r		114
	a		97

		z		122
		e		101
"zebra"	==	b	==	98
		r		114
		a		97
				532



"zebra" ==

z
e
b
r
a

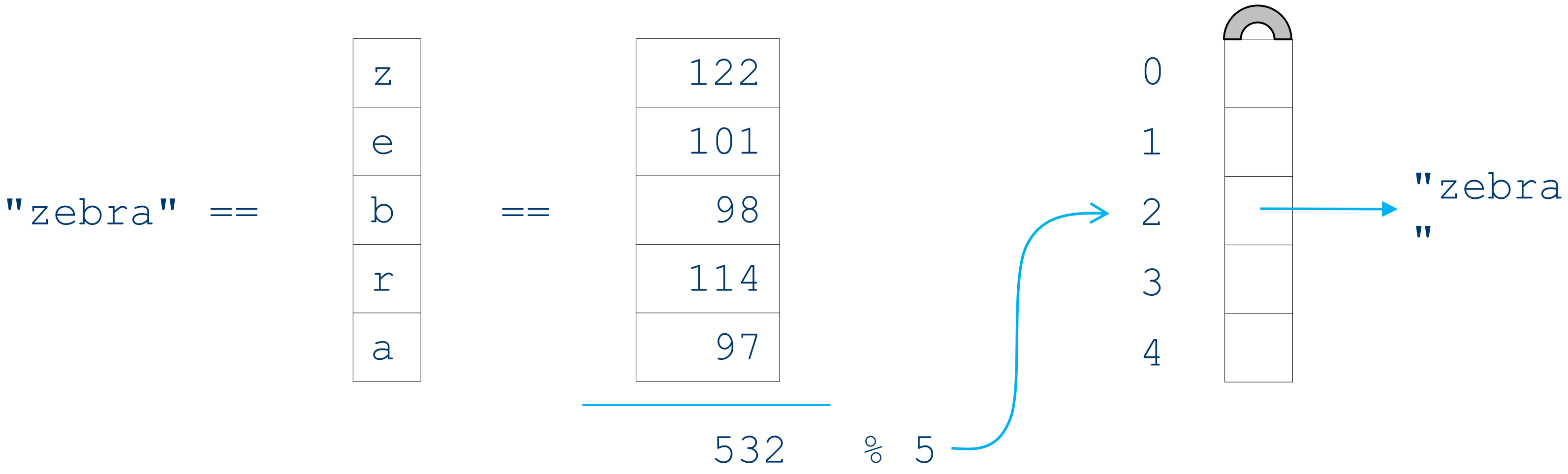
==

122
101
98
114
97

532 % 5

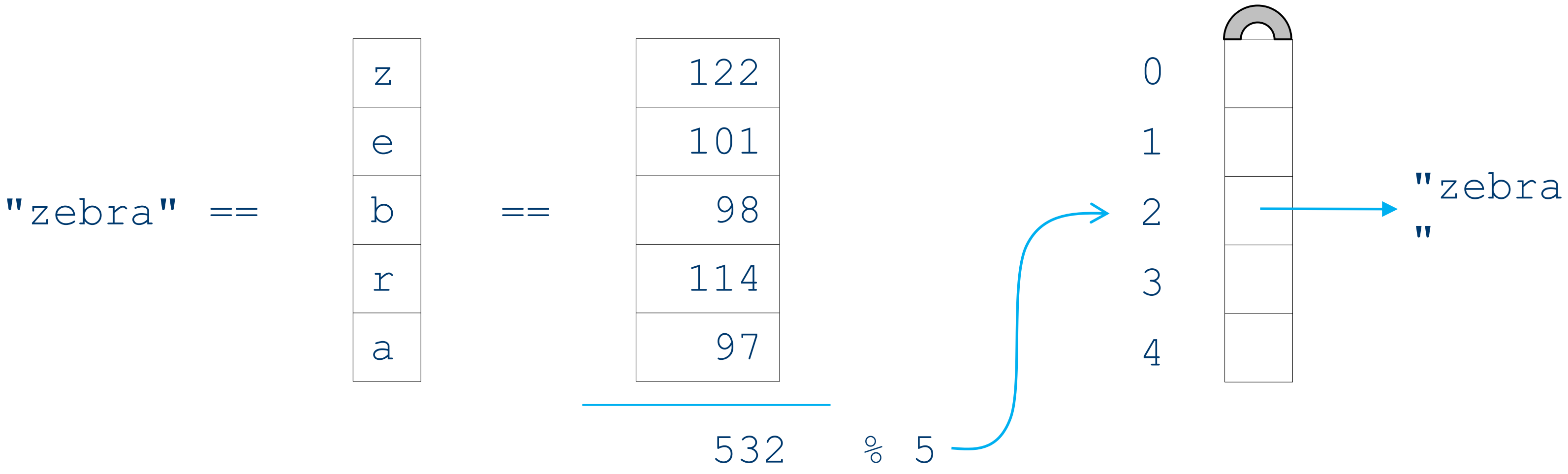
0
1
2
3
4

"zebra
"



If we can define a hash function for something,

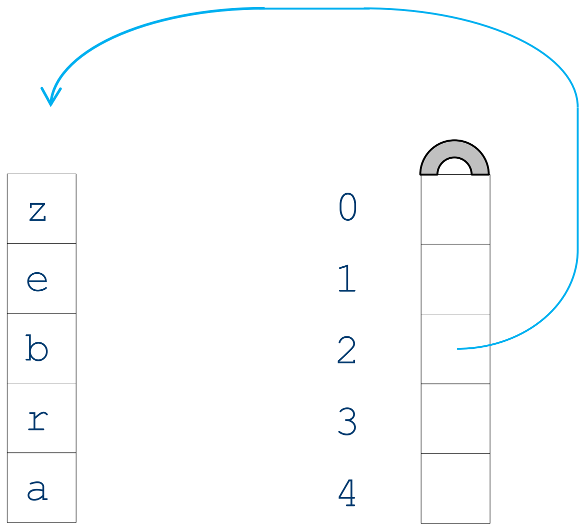
we can store it in a set



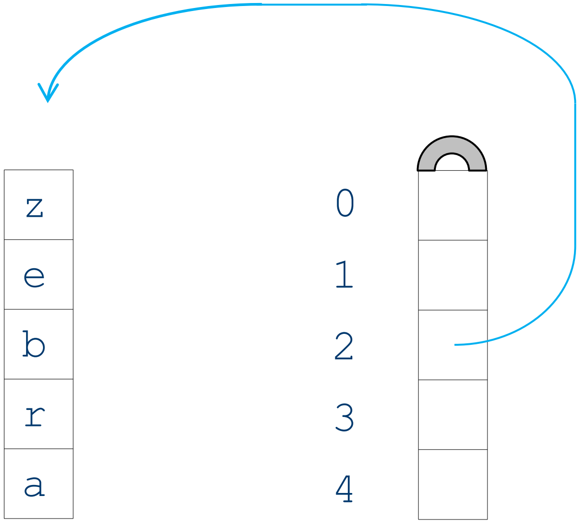
If we can define a hash function for something,

we can store it in a set

So long as nothing changes behind our back



This is what the previous example really looks like in memory

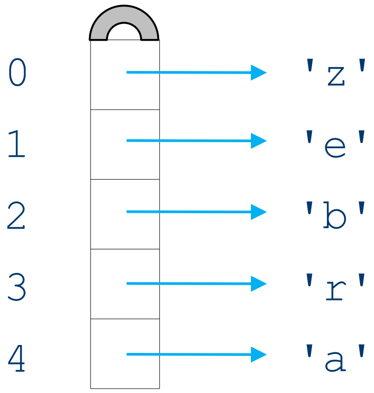


This is what the previous example really looks like in memory

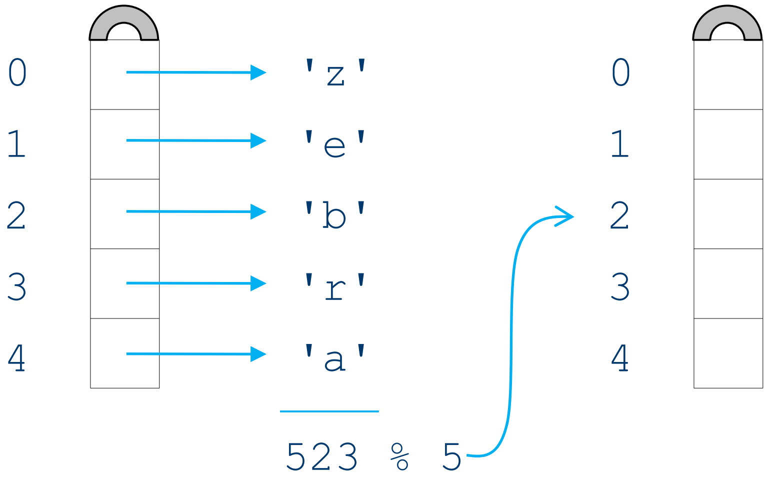
Let's take a look at what happens if we use a list

```
['z','e','b','r','a']
```

```
['z','e','b','r','a'] ==
```



['z','e','b','r','a'] ==



0 'z'

1 'e'

2 'b'

3 'r'

4 'a'

523 % 5

0

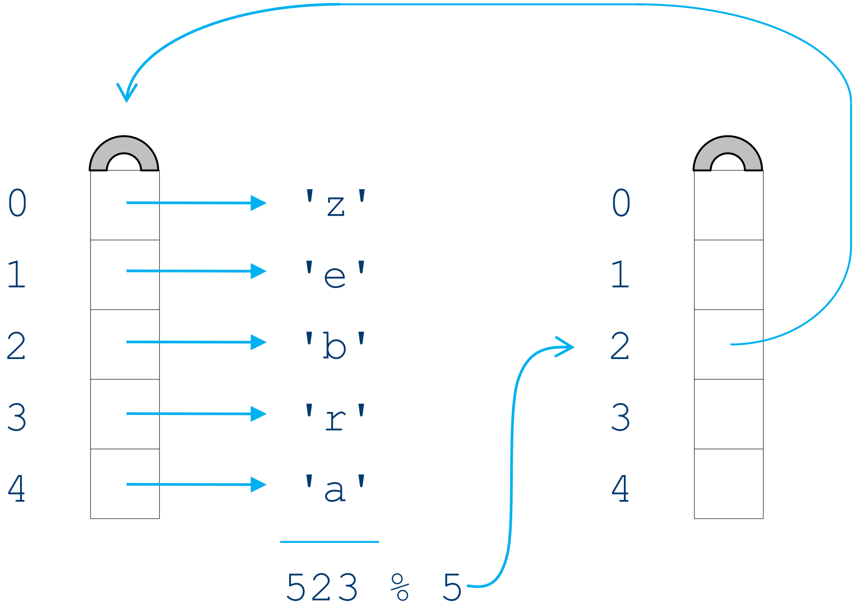
1

2

3

4

['z','e','b','r','a'] ==



0 'z' 0

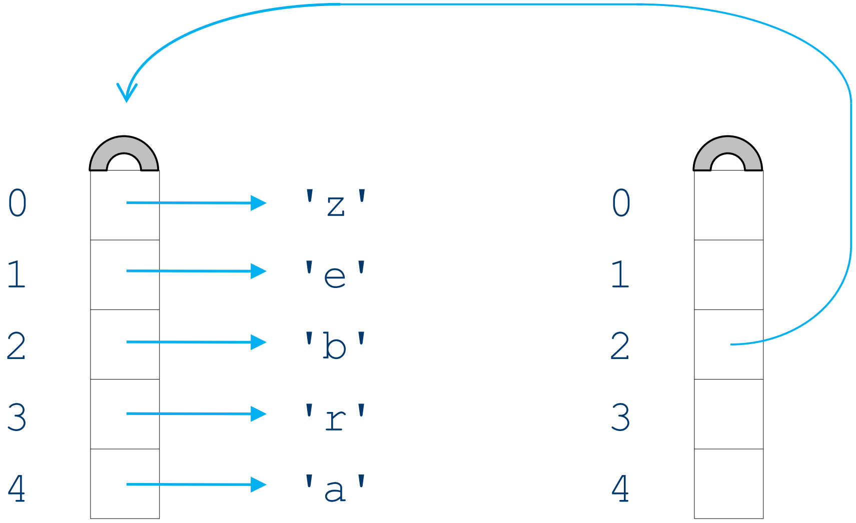
1 'e' 1

2 'b' 2

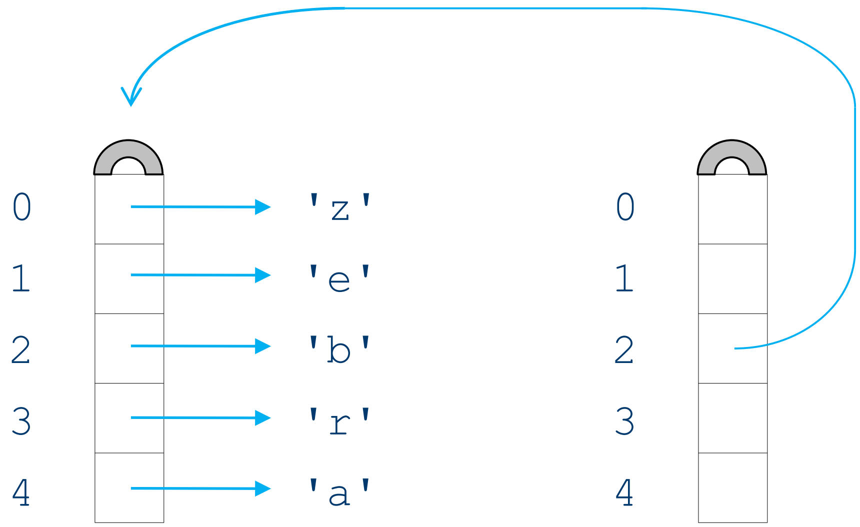
3 'r' 3

4 'a' 4

523 % 5

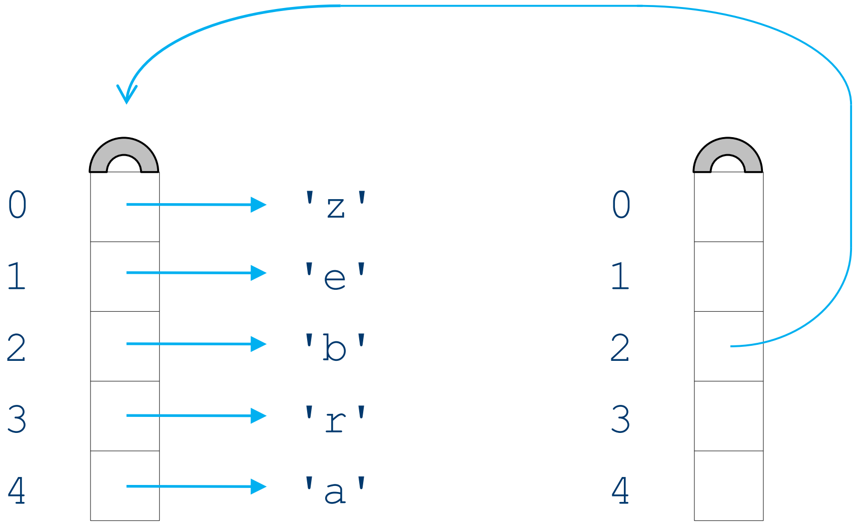


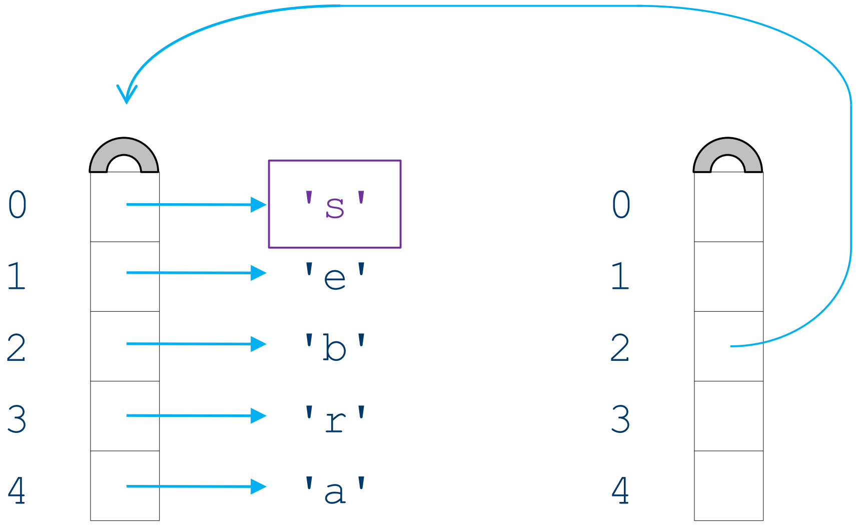
This is what's actually in memory

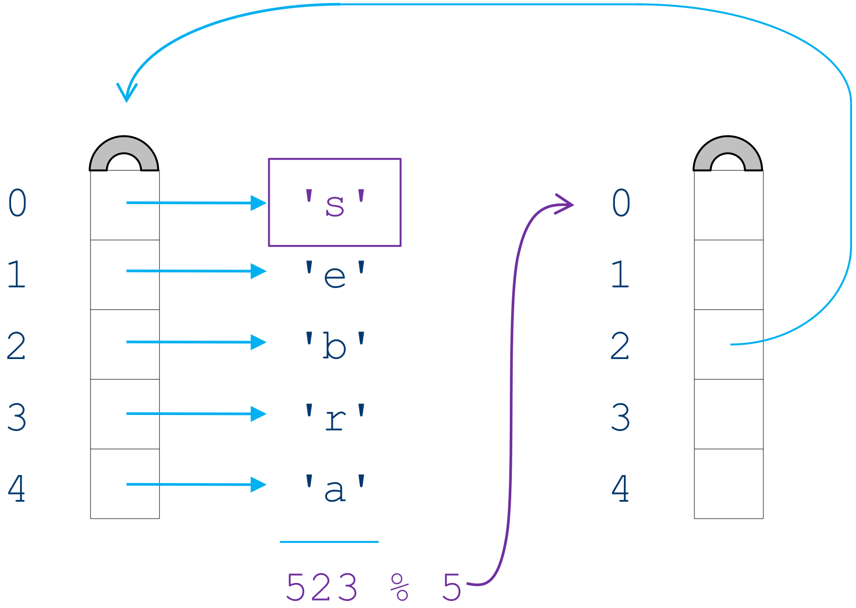


This is what's actually in memory

What happens if we change the values in the list?

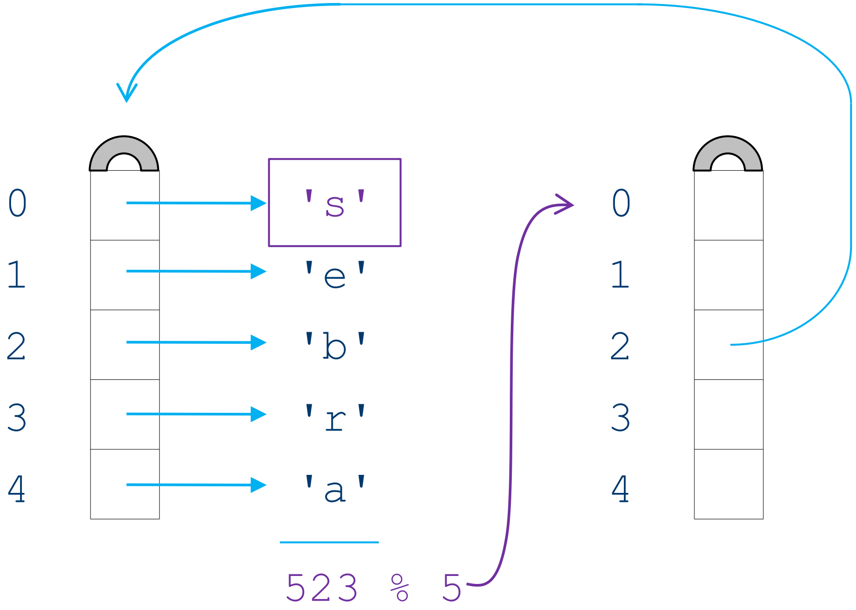

0
1
2
3
4
'z'
'e'
'b'
'r'
'a'
0
1
2
3
4


0
1
2
3
4
's'
'e'
'b'
'r'
'a'
0
1
2
3
4

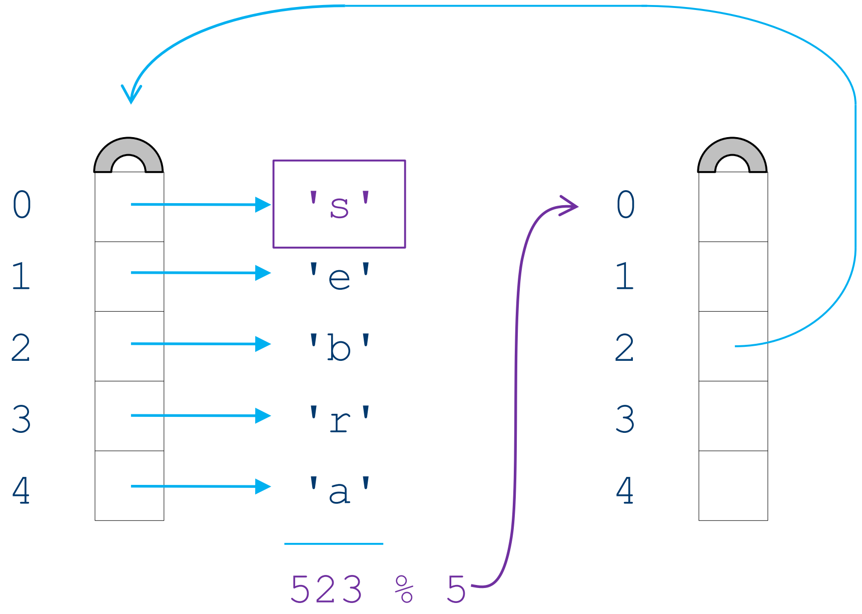


0
1
2
3
4
's'
'e'
'b'
'r'
'a'
523 % 5
0
1
2
3
4

The list is stored in the wrong place!

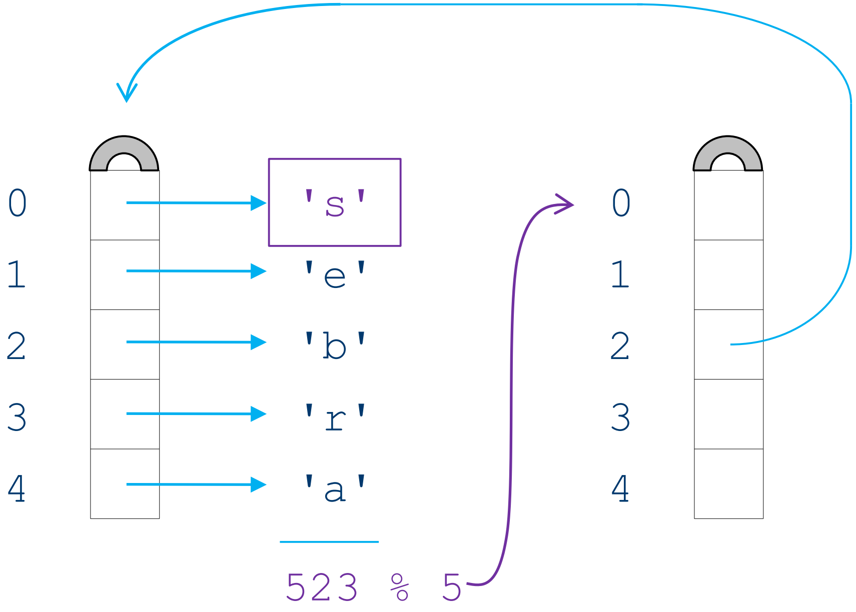


The list is stored in the wrong place!



```
['s','e','b','r','a'] in S
```

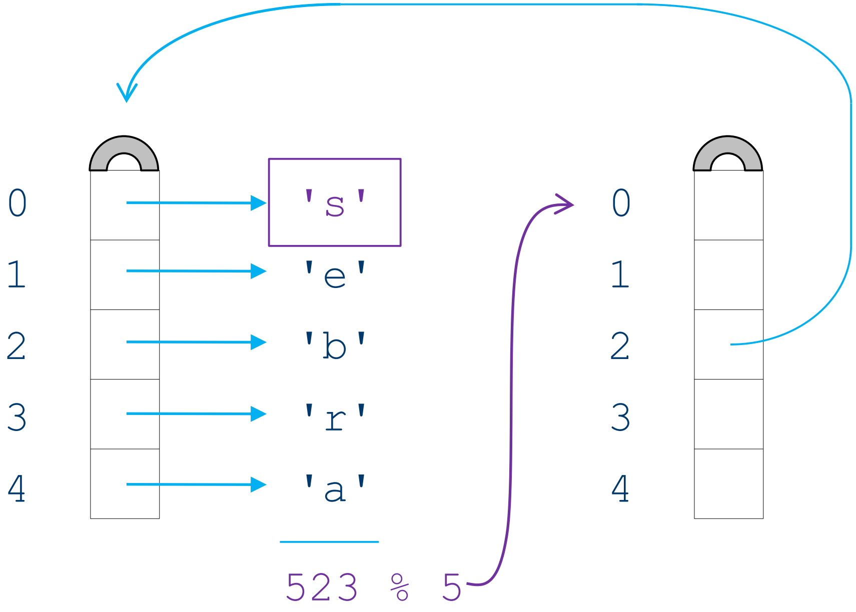
The list is stored in the wrong place!



`['s','e','b','r','a'] in S`

looks at index `0` and says `False`

The list is stored in the wrong place!

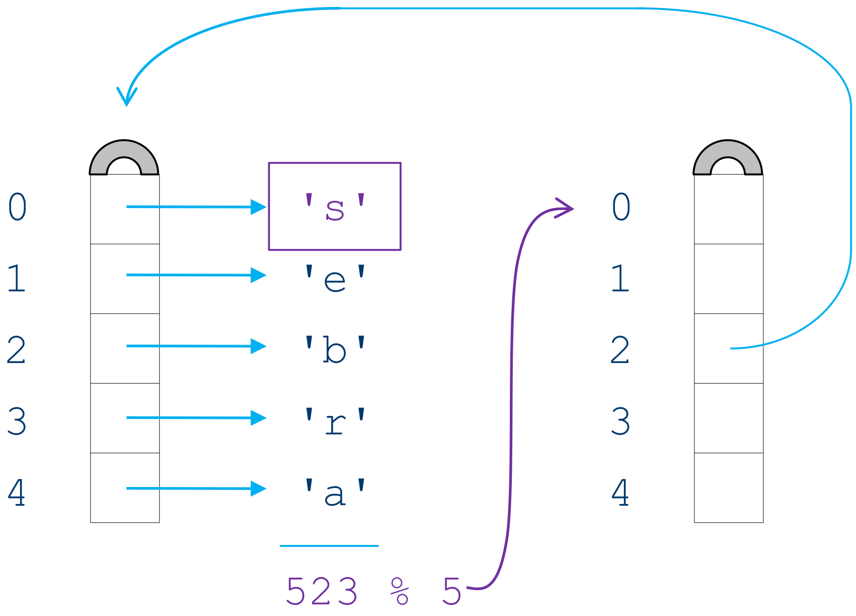


`['s','e','b','r','a'] in S`

looks at index 0 and says `False`

`['z','e','b','r','a'] in S`

The list is stored in the wrong place!



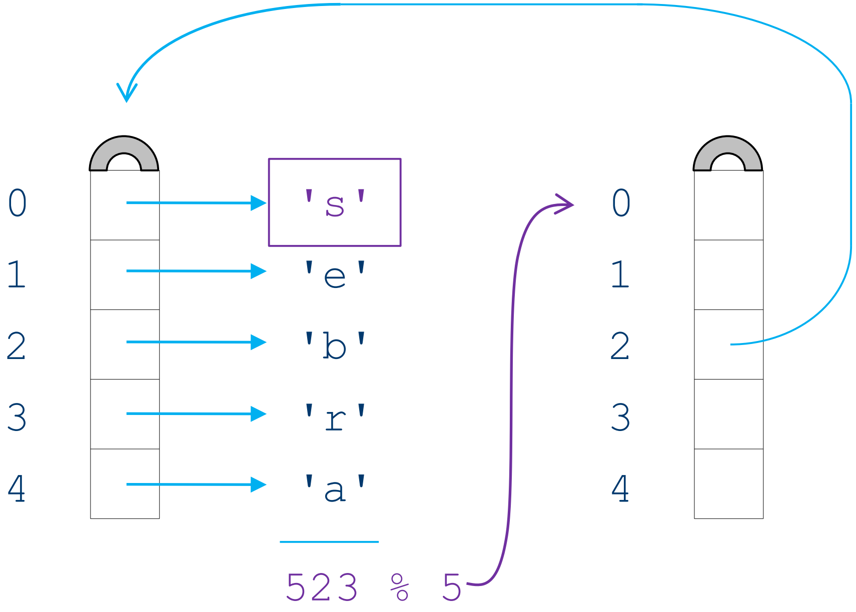
`['s','e','b','r','a'] in S`

looks at index 0 and says `False`

`['z','e','b','r','a'] in S`

looks at index 2 and says True

The list is stored in the wrong place!



`['s','e','b','r','a'] in S`

looks at index 0 and says `False`

`['z','e','b','r','a'] in S`

looks at index 2 and says True (or blows up)

This problem arises with any *mutable* structure

This problem arises with any *mutable* structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

This problem arises with any mutable structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

Very expensive

This problem arises with any mutable structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

Very expensive

Option #2: allow it, and blame the programmer

This problem arises with any mutable structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

Very expensive

Option #2: allow it, and blame the programmer

Very expensive when it goes wrong

UNIVERSITY
OF ONTARIO
INSTITUTE OF TECHNOLOGY

This problem arises with any mutable structure

Option #1: keep track of the sets an object is in, and update pointers every time the object changes

Very expensive

Option #2: allow it, and blame the programmer

Very expensive when it goes wrong

Option #3: only permit *immutable* objects in sets

This problem arises with any mutable structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

Very expensive

Option #2: allow it, and blame the programmer

Very expensive when it goes wrong

Option #3: only permit immutable objects in sets

(If an object can't change, neither can its hash value)

This problem arises with any mutable structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

Very expensive

Option #2: allow it, and blame the programmer

Very expensive when it goes wrong

Option #3: only permit immutable objects in sets

(If an object can't change, neither can its hash value)

Slightly restrictive, but never disastrous

So how do we store values that naturally have several parts, like first name and last name?

So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

'Charles' and 'Darwin' stored as 'Charles|Darwin'

So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

'Charles' and 'Darwin' stored as 'Charles|Darwin'

(Can't use space to join 'Paul Antoine' and 'St. Cyr')

So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

'Charles' and 'Darwin' stored as 'Charles|Darwin'

(Can't use space to join 'Paul Antoine' and 'St. Cyr')

But data *always* changes...

So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

'Charles' and 'Darwin' stored as 'Charles|Darwin'

(Can't use space to join 'Paul Antoine' and 'St. Cyr')

But data always changes...

Code has to be littered with joins and splits

Option #2 (in Python): use a tuple

Option #2 (in Python): use a tuple

An immutable list

Option #2 (in Python): use a tuple

An immutable list

Contents cannot be changed after tuple is created

```
>>> full_name = ('Charles', 'Darwin')
>>> full_name = ('Charles', 'Darwin')
```

Use '()' instead of '[]'

```
>>> full_name = ('Charles', 'Darwin')
>>> full_name[0]
Charles
```

```
>>> full_name = ('Charles', 'Darwin')
>>> full_name[0]
Charles

>>> full_name[0] = 'Erasmus'
TypeError: 'tuple' object does not support item
assignment
```

```
>>> full_name = ('Charles', 'Darwin)
>>> full_name[0]
Charles

>>> full_name[0] = 'Erasmus'
TypeError: 'tuple' object does not support item
assignment

>>> names = set()
>>> names.add(full_name)
>>> names
set([('Charles', 'Darwin')])
```

This part has been about the science of computer science

This part has been about the science of computer science

- Designs for hash tables

This part has been about the science of computer science

- Designs for hash tables

- Mutability, usability, and performance

This part has been about the science of computer science

- Designs for hash tables
- Mutability, usability, and performance

It's a lot to digest in one go...

This part has been about the science of computer science

- Designs for hash tables

- Mutability, usability, and performance

It's a lot to digest in one go...

...but sometimes you need a little theory to make sense of practice

Python

Sets and Dictionaries

Dictionaries

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Back to the data from our summer counting birds in a mosquito-infested swamp in northern Ontario

Back to the data from our summer counting birds in a mosquito-infested swamp in northern Ontario

How many birds of each kind did we see?

Back to the data from our summer counting birds in a mosquito-infested swamp in northern Ontario

How many birds of each kind did we see?

Input is a list of several thousand bird names

Back to the data from our summer counting birds in a mosquito-infested swamp in northern Ontario

How many birds of each kind did we see?

Input is a list of several thousand bird names

Output is a list of names and counts

Could use a list of [name, count] pairs

Could use a list of [name, count] pairs

```
def another_bird(counts, bird_name):
  for i in range(len(counts)):
    if counts[i][0] == bird_name:
      counts[i][1] += 1
      return
  counts.append([bird_name, 1])
```

Could use a list of [name, count] pairs

```
def another_bird(counts, bird_name):
  for i in range(len(counts)):
    if counts[i][0] == bird_name:
      counts[i][1] += 1
      return
  counts.append([bird_name, 1])
```

List of pairs

Could use a list of [name, count] pairs

```
def another_bird(counts, bird_name):
  for i in range(len(counts)):
    if counts[i][0] == bird_name:
      counts[i][1] += 1
      return
  counts.append([bird_name, 1])
```

Name to add

Could use a list of [name, count] pairs

```
def another_bird(counts, bird_name):
  for i in range(len(counts)):
    if counts[i][0] == bird_name:
      counts[i][1] += 1
      return
  counts.append([bird_name, 1])
```

Look at each pair already in the list

Could use a list of [name, count] pairs

```
def another_bird(counts, bird_name):
  for i in range(len(counts)):
    if counts[i][0] == bird_name:
      counts[i][1] += 1
      return
  counts.append([bird_name, 1])
```

If this is the bird we're looking for...

Could use a list of [name, count] pairs

```
def another_bird(counts, bird_name):
  for i in range(len(counts)):
    if counts[i][0] == bird_name:
      counts[i][1] += 1
      return
  counts.append([bird_name, 1])
```

...add 1 to its

count and finish

Could use a list of [name, count] pairs

```
def another_bird(counts, bird_name):
  for i in range(len(counts)):
    if counts[i][0] == bird_name:
      counts[i][1] += 1
      return
  counts.append([bird_name, 1])
```

Otherwise, add

a new pair to the list

Could use a list of [name, count] pairs

```
def another_bird(counts, bird_name):
  for i in range(len(counts)):
    if counts[i][0] == bird_name:
      counts[i][1] += 1
      return
  counts.append([bird_name, 1])
```

Pattern: handle an existing case and return in loop,

or take default action if we exit the loop normally

Could use a list of [name, count] pairs

```
def another_bird(counts, bird_name):
  for i in range(len(counts)):
    if counts[i][0] == bird_name:
      counts[i][1] += 1
      return
  counts.append([bird_name, 1])
```

```
start                       []
```

Could use a list of [name, count] pairs

```
def another_bird(counts, bird_name):
  for i in range(len(counts)):
    if counts[i][0] == bird_name:
      counts[i][1] += 1
      return
  counts.append([bird_name, 1])
```

```
start                   []
loon                    [['loon', 1]]
```

Could use a list of [name, count] pairs

```
def another_bird(counts, bird_name):
  for i in range(len(counts)):
    if counts[i][0] == bird_name:
      counts[i][1] += 1
      return
  counts.append([bird_name, 1])
```

start	[]
loon	[['loon', 1]]
goose	[['loon', 1], ['goose', 1]]

Could use a list of [name, count] pairs

```
def another_bird(counts, bird_name):
  for i in range(len(counts)):
    if counts[i][0] == bird_name:
      counts[i][1] += 1
      return
  counts.append([bird_name, 1])
```

```
start                   []
loon                    [['loon', 1]]
goose                   [['loon', 1], ['goose', 1]]
loon                    [['loon', 2], ['goose', 1]]
```

There's a better way

There's a better way

Use a *dictionary*

There's a better way

Use a *dictionary*

An unordered collection of key/value pairs

There's a better way

Use a *dictionary*

An unordered collection of key/value pairs

Like set elements, keys are:

There's a better way

Use a *dictionary*

An unordered collection of key/value pairs

Like set elements, keys are:

- Immutable

There's a better way

Use a *dictionary*

An unordered collection of key/value pairs

Like set elements, keys are:

- Immutable

- Unique

There's a better way

Use a *dictionary*

An unordered collection of key/value pairs

Like set elements, keys are:

- Immutable
- Unique
- Not stored in any particular order

There's a better way

Use a *dictionary*

An unordered collection of key/value pairs

Like set elements, keys are:

- Immutable
- Unique
- Not stored in any particular order

No restrictions on values

There's a better way

Use a *dictionary*

An unordered collection of key/value pairs

Like set elements, keys are:

- Immutable
- Unique
- Not stored in any particular order

No restrictions on values

- Don't have to be immutable or unique

Create a dictionary by putting `key:value` pairs in `{}`

Create a dictionary by putting `key:value` pairs in `{}`

```
>>> birthdays = {'Newton' : 1642, 'Darwin' : 1809}
```

Create a dictionary by putting `key:value` pairs in `{}`

```
>>> birthdays = {'Newton' : 1642, 'Darwin' : 1809}
```

Retrieve values by putting key in []

Create a dictionary by putting `key:value` pairs in `{}`

```
>>> birthdays = {'Newton' : 1642, 'Darwin' : 1809}
```

Retrieve values by putting key in []

Just like indexing strings and lists

Create a dictionary by putting `key:value` pairs in `{}`

```
>>> birthdays = {'Newton' : 1642, 'Darwin' : 1809}
```

Retrieve values by putting key in []

Just like indexing strings and lists

```
>>> print(birthdays['Newton'])
1642
```

Create a dictionary by putting `key:value` pairs in { }

```
>>> birthdays = {'Newton' : 1642, 'Darwin' : 1809}
```

Retrieve values by putting key in []

Just like indexing strings and lists

```
>>> print(birthdays['Newton'])
1642
```

Just like using a phonebook or dictionary

Add another value by assigning to it

Add another value by assigning to it

```
>>> birthdays['Turing'] = 1612    # that's not right
```

Add another value by assigning to it

```
>>> birthdays['Turing'] = 1612    # that's not right
```

Overwrite value by assigning to it as well

Add another value by assigning to it

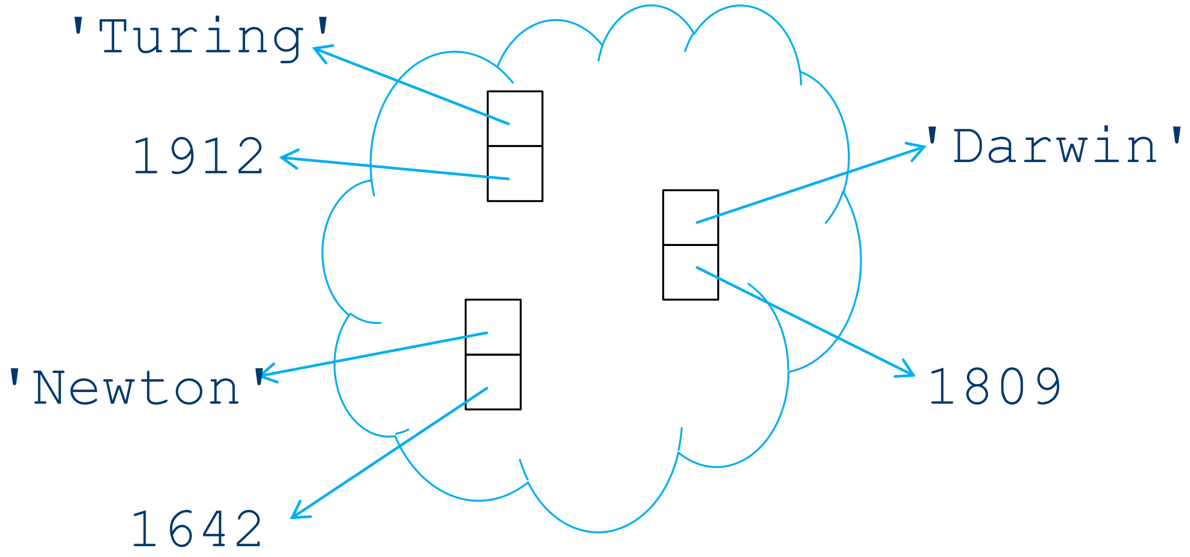
```
>>> birthdays['Turing'] = 1612    # that's not right
```

Overwrite value by assigning to it as well

```
>>> birthdays['Turing'] = 1912
>>> print(birthdays)
{'Turing' : 1912, 'Newton' : 1642, 'Darwin' : 1809}
```

Note: entries are *not* in any particular order

Note: entries are *not* in any particular order



Key must be in dictionary *before* use

Key must be in dictionary *before* use

```
>>> birthdays['Nightingale']
KeyError: 'Nightingale'
```

Key must be in dictionary *before* use

```
>>> birthdays['Nightingale']
KeyError: 'Nightingale'
```

Test whether key is present using `in`

Key must be in dictionary *before* use

```
>>> birthdays['Nightingale']
KeyError: 'Nightingale'
```

Test whether key is present using `in`

```
>>> 'Nightingale' in birthdays
False
>>> 'Darwin' in birthdays
True
```

Use `for` to loop over keys

Use `for` to loop over keys

Unlike lists, where `for` loops over values

Use for to loop over keys

Unlike lists, where for loops over values

```
>>> for name in birthdays:
...   print(name, birthdays[name])

Turing 1912
Newton 1642
Darwin 1809
```

Let's count those birds

Let's count those birds

```
import sys

if __name__ == '__main__':
  reader = open(sys.argv[1], 'r')
  lines = reader.readlines()
  reader.close()
  count = count_names(lines)
  for name in count:
    print(name, count[name])
```

Let's count those birds

```
import sys

if __name__ == '__main__':
  reader = open(sys.argv[1], 'r')
  lines = reader.readlines()
  reader.close()
  count = count_names(lines)
  for name in count:
    print(name, count[name])
```

Read all the data

Let's count those birds

```
import sys

if __name__ == '__main__':
  reader = open(sys.argv[1], 'r')
  lines = reader.readlines()
  reader.close()
  count = count_names(lines)
  for name in count:
    print(name, count[name])
```

← Count distinct values

Let's count those birds

```
import sys

if __name__ == '__main__':
  reader = open(sys.argv[1], 'r')
  lines = reader.readlines()
  reader.close()
  count = count_names(lines)
  for name in count:
    print(name, count[name])
```

← Show results

```python
def count_names(lines):
 '''Count unique lines of text, returning dictionary.'''

  result = {}
  for name in lines:
    name = name.strip()
    if name in result:
      result[name] = result[name] + 1
    else:
      result[name] = 1

  return result
```

```
def count_names(lines):
 '''Count unique lines of text, returning dictionary.'''

  result = {}
  for name in lines:
    name = name.strip()
    if name in result:
      result[name] = result[name] + 1
    else:
      result[name] = 1

  return result
```

Explain what we're doing to the next reader

```
def count_names(lines):
 '''Count unique lines of text, returning dictionary.'''

  result = {}
  for name in lines:
    name = name.strip()
    if name in result:
      result[name] = result[name] + 1
    else:
      result[name] = 1

  return result
```

Create an empty dictionary to fill

```
def count_names(lines):
 '''Count unique lines of text, returning dictionary.'''

  result = {}
  for name in lines:
    name = name.strip()
    if name in result:
      result[name] = result[name] + 1
    else:
      result[name] = 1

  return result
```

Handle input values one at a time

```
def count_names(lines):
 '''Count unique lines of text, returning dictionary.'''

  result = {}
  for name in lines:
    name = name.strip()
    if name in result:
      result[name] = result[name] + 1
    else:
      result[name] = 1

  return result
```

Clean up before processing

```python
def count_names(lines):
  '''Count unique lines of text, returning dictionary.'''

  result = {}
  for name in lines:
    name = name.strip()
    if name in result:
      result[name] = result[name] + 1
    else:
      result[name] = 1

  return result
```

If we have seen this value before…

```
def count_names(lines):
  '''Count unique lines of text, returning dictionary.'''

  result = {}
  for name in lines:
    name = name.strip()
    if name in result:
      result[name] = result[name] + 1
    else:
      result[name] = 1

  return result
```

← add one to its count

```
def count_names(lines):
 '''Count unique lines of text, returning dictionary.'''

  result = {}
  for name in lines:
    name = name.strip()
    if name in result:
      result[name] = result[name] + 1
    else:
      result[name] = 1

  return result
```

But if it's the first time we have seen this name, store it with a count of 1

```
def count_names(lines):
 '''Count unique lines of text, returning dictionary.'''

  result = {}
  for name in lines:
    name = name.strip()
    if name in result:
      result[name] = result[name] + 1
    else:
      result[name] = 1

  return result
```

Return the result

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Counter in action

Counter in action

start	{ }

Counter in action

start	`{}`
`loon`	`{'loon' : 1}`

Counter in action

start	`{}`
`loon`	`{'loon' : 1}`
`goose`	`{'loon' : 1, 'goose' : 1}`

Counter in action

start	`{}`
`loon`	`{'loon' : 1}`
`goose`	`{'loon' : 1, 'goose' : 1}`
`loon`	`{'loon' : 2, 'goose' : 1}`

Counter in action

```
start                     {}
loon                      {'loon' : 1}
goose                     {'loon' : 1, 'goose' : 1}
loon                      {'loon' : 2, 'goose' : 1}
```

But like sets, dictionaries are much more efficient than lookup lists