

# Dictionaries

---



## CS111 Computer Programming

Department of Computer Science  
Wellesley College

# Looking up English words in the dictionary

**Concepts in this slide:**  
Comparing sequences to  
collections.

**Sequence** : a group of things  
that come one after the other



**Collection** : a group of (interesting)  
things brought together for some  
purpose



# Looking up English words in the dictionary

**Concepts in this slide:**  
Comparing sequences to  
collections.

**Sequence** : a group of things  
that come one after the other



Is a sequence a collection?

**Collection** : a group of (interesting)  
things brought together for some  
purpose



Is a collection a sequence?

# Looking up English words in the dictionary

**Concepts in this slide:**  
Comparing sequences to collections.

**Sequence** : a group of things that come one after the other



**Collection** : a group of (interesting) things brought together for some purpose



Is a sequence a collection? **Yes!**

Is a collection a sequence? **No.**

- A **sequence** is an ordered collection in which elements can be accessed by index. All sequences are collections but not all collections are sequences.

# Properties of sequences and collections

**Concepts in this slide:**  
Properties that are common and distinct for the two categories.

- Collections
  - Find their length with **len**
  - Check an element membership in the collection with **in**
  - Are iterables (one can iterate over their elements with a loop)
- Sequences
  - Use indices to access elements, e.g. **myList[2]**
  - Use slice operations to access subsequences, e.g. **myList[2:5]**
- Mutable: can be changed through object methods.
- Immutable: cannot be changed.

# Python collections

**Concepts in this slide:**  
Definitions and examples  
of Python collection types.

Type	Description	Example
list	a mutable <u>sequence</u> of arbitrary objects	<code>[-100, "blue", (1, 10), True]</code>
tuple	an immutable <u>sequence</u> of arbitrary objects	<code>(2017, "Mar", 2)</code>
string	an immutable <u>sequence</u> of characters	<code>"Go Wellesley!"</code>
range	an immutable <u>sequence</u> of numbers	<code>range(3)</code>
set	a mutable unordered <u>collection</u> of distinct objects.	<code>{1, 4, 5, 23}</code>
dict	a mutable unordered <u>collection</u> of key:value pairs, where keys are <b>immutable</b> and values are any Python objects	<code>{"orange": "fruit", 3: "March", "even": [2, 4, 6, 8]}</code>


# Dictionaries

**Concepts in this slide:**  
New type: dictionary, its syntax (use { }), and key:value pairs.

A Python **dictionary** is a **mutable** collection that maps **keys** to **values**.

A **dictionary** is enclosed with curly brackets and contains comma-separated pairs. A **pair** is a **colon-separated** key and value.

```
daysInMonth = { 'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30, ... }
```



```
monthLengths = { 31: [ 'Jan', 'Mar', 'May', 'Jul', 'Aug', 'Oct', 'Dec' ],  
                 30: [ 'Apr', 'Jun', 'Sep', 'Nov' ],  
                 28: [ 'Feb' ]  
 }
```



# keys cannot mutate

**Concepts in this slide:**  
An analogy to P.O. box keys. If the key is damaged, one cannot retrieve the content.





# keys

## Concepts in this slide:

Keys can only be numbers, strings, or tuples.

**keys:** any *immutable* type such as numbers, strings, or tuples.

```
phones = {5558671234: 'Gal Gadot',  
          9996541212: 'Trevor Noah',  
          7811234567: 'Paula A. Johnson'}
```

```
daysInMonth = {'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30,  
               'May': 31, 'Jun': 30, 'Jul': 31, 'Aug': 31,  
               'Sep': 30, 'Oct': 30, 'Nov': 30}
```

```
heroes = {('Diana', 'Prince'): ['ww@dc-comics.com', 'Wonderwoman'],  
          ('Peter', 'Parker'): ['sm@marvel.com', 'Spiderman'],  
          ('Clark', 'Kent'): ['sm@dc-comics.com', 'Superman']}
```

## To notice:

In `daysInMonth`, the key for December is missing, it will be added later in the slides.

# values

**Concepts in this slide:**  
Differently from keys, a value can be any object.

**values:** *any* Python object (numbers, strings, lists, tuples, dicts, sets, even functions)

```
student = { 'name': 'Georgia Dome', 'dorm': 'Munger Hall',  
            'year': 2019, 'CSMajor?': True }
```

```
townNames = { 'MA': ['Boston', 'Worcester', 'Springfield'],  
              'CT': ['Hartford', 'Danbury', 'New Haven'] }
```

```
contributions = { 'uma52': {2015: 10, 2016: 15},  
                  'setam$3': {2012: 23, 2013: 34, 2014: 17},  
                  'rid12': {2009: 5, 2010: 18, 2012: 4}  
                }
```

# How do we create dictionaries?

**Concepts in this slide:**  
Three common ways to create dictionaries.

1. Literal dictionary: provide keys and pairs together, delimited with { }

```
In [1]: scrabbleDict = {'a': 1, 'b': 3, 'c': 3, 'd': 2, 'e': 1, 'f': 4, 'g': 2, 'h': 4, 'i': 1, 'j': 8, 'k': 5, 'l': 1, 'm': 3, 'n': 1, 'o': 1, 'p': 3, 'q': 10, 'r': 1, 's': 1, 't': 1, 'u': 1, 'v': 4, 'w': 4, 'x': 8, 'y': 4, 'z': 10}
```

2. Start with an empty dict and add key/pairs

```
In [2]: cart = {} # an empty dict
In [3]: cart['oreos'] = 3.99
In [4]: cart['kiwis'] = 2.54
In [5]: cart
Out[5]: {'kiwis': 2.54, 'oreos': 3.99}
```

3. Applying the built-in function **dict** to a list of tuples:

```
In [6]: dict([('DEU', 49), ('ALB', 355), ('UK', 44)])
Out[6]: {'ALB': 355, 'DEU': 49, 'UK': 44}
```

## To notice:

The output Out[5] doesn't show the items in the dictionary in the same order they were added. Never except that items will be ordered.

# Dictionary Operations: subscripting

**Concepts in this slide:**  
We use the subscripting notation with key(s) to access values.

The **value** associated with a **key** is accessed using the same subscripting notation with square brackets used for list indexing:

```
In [7]: daysInMonth['Oct']
```

```
Out[7]: 31
```

```
In [8]: heroes[('Peter', 'Parker')]
```

```
Out[8]: ['sm@marvel.com', 'Spiderman']
```

```
In [9]: phones[5558671234]
```

```
Out[9]: 'Gal Gadot'
```

```
In [10]: townNames['CT'][2]
```

```
Out[10]: 'New Haven'
```

```
In [11]: contributions['rid12'][2010]
```

```
Out[11]: 18
```

key      list index

key      key

## To notice:

Inputs [10] and [11] use double subscription to access elements that are nested within complex values. In In[10], an element within a list, in In[11] a value within a nested dictionary.

# Dictionary Operations:

## check with **in** before accessing

### Concepts in this slide:

Non-existing keys raise an error, use the operator **in** to check if key exists.

Subscripting a dictionary with an invalid key raises a `KeyError`:

```
In [12]: daysInMonth['October']

-----

KeyError
Traceback (most recent call last)
<ipython-input-4-3d32324d55ec>
in <module>()
----> 1 daysInMonth['October']

KeyError: 'October'
```

One way to avoid such errors is to use **in** to check if a key exists

```
In [13]: 'Oct' in daysInMonth
Out[13]: True

In [14]: 'October' in daysInMonth
Out[14]: False
```

# Mutability in Dictionaries

## Dictionaries are mutable

remember the variable `daysOfMonth`?

```
daysInMonth = { 'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30,  
                'May': 31, 'Jun': 30, 'Jul': 31, 'Aug': 31,  
                'Sep': 30, 'Oct': 30, 'Nov': 30 }
```

- We can add or remove key-value pairs
- We can change the value associated with a key

```
daysInMonth[ 'Feb' ] = 29    # change for leap year  
daysInMonth[ 'Dec' ] = 31    # add new key and value
```

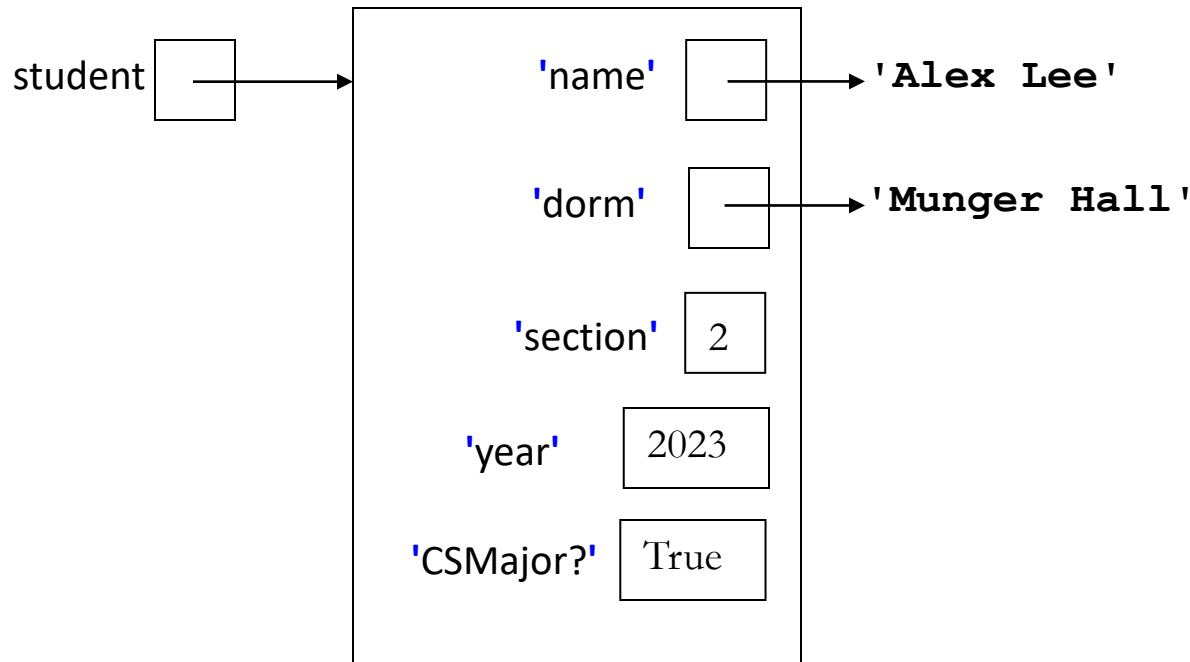
## Reminder: dictionary keys must be immutable

Eg: a list or a dict cannot be a key (only immutable values such as numbers, strings, and tuples)



# Memory diagram for a dictionary

```
student = { 'name': 'Alex Lee', 'dorm': 'Munger Hall',  
            'section': 2, 'year': 2023, 'CSMajor?': True }
```



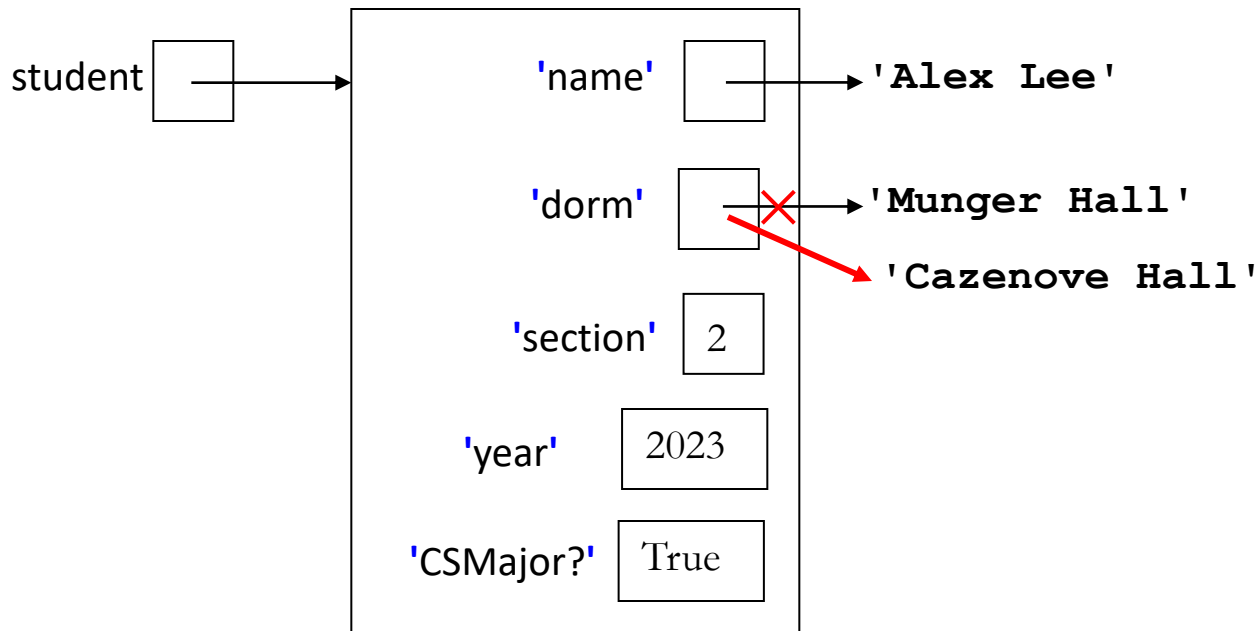
## Memory Diagram note:

Only primitive values such as numbers, Booleans, and None are displayed within the slots. Other values (strings, tuples, lists, objects) are shown outside.

Notice how the “keys” are shown similarly to the indices in sequences. However, they are **not ordered** in any meaningful way.

# Dictionaries are Mutable: change value for key

The value associated with a key can be changed by combining subscript and assignment notation:



```
In [16]: student['dorm'] = 'Cazenove Hall'
```

```
In [17]: student
```

```
Out[17]: {'CSMajor?': True, 'dorm': 'Cazenove Hall',  
'name': 'Alex Lee', 'section': 2, 'year': 2023}
```

## Concepts in this slide:

An assignment statement is used to add new key/value pairs or change existing ones.

## To notice:

Canopy displays dictionaries with string-valued keys in ASCII order (see Out[22]). But, this is not true in other environments, thus, don't rely on order or keys.

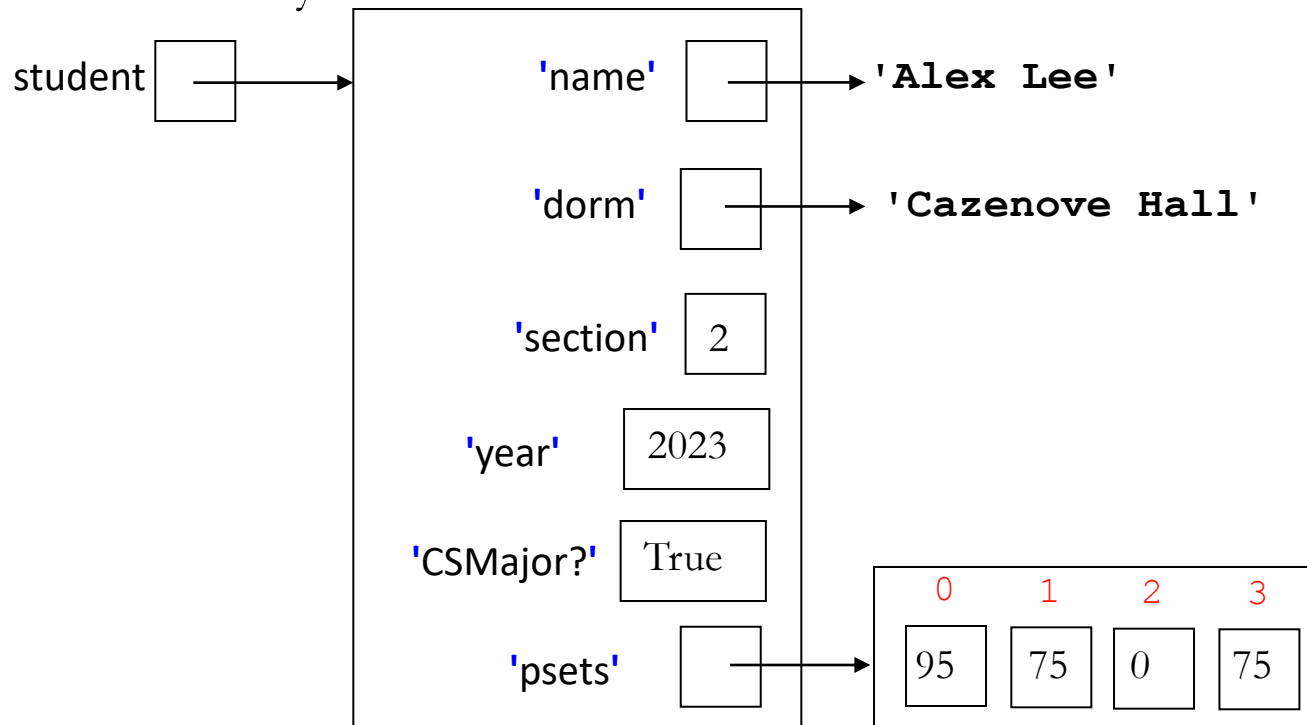
# Dictionaries are Mutable:

## add key/value pair

### Concepts in this slide:

Adding key/value pairs is the most common way to create and change dicts.

A new *key/value* pair can be added by assigning to a key not already in the dictionary:



```
In [18]: student['psets'] = [95, 75, 0, 75]
```

```
In [19]: student
```

```
Out[19]: {'CSMajor?': True, 'dorm': 'Cazenove Hall',  
'name': 'Alex Lee', 'psets': [95, 75, 0, 75],  
'section': 2, 'year': 2023}
```

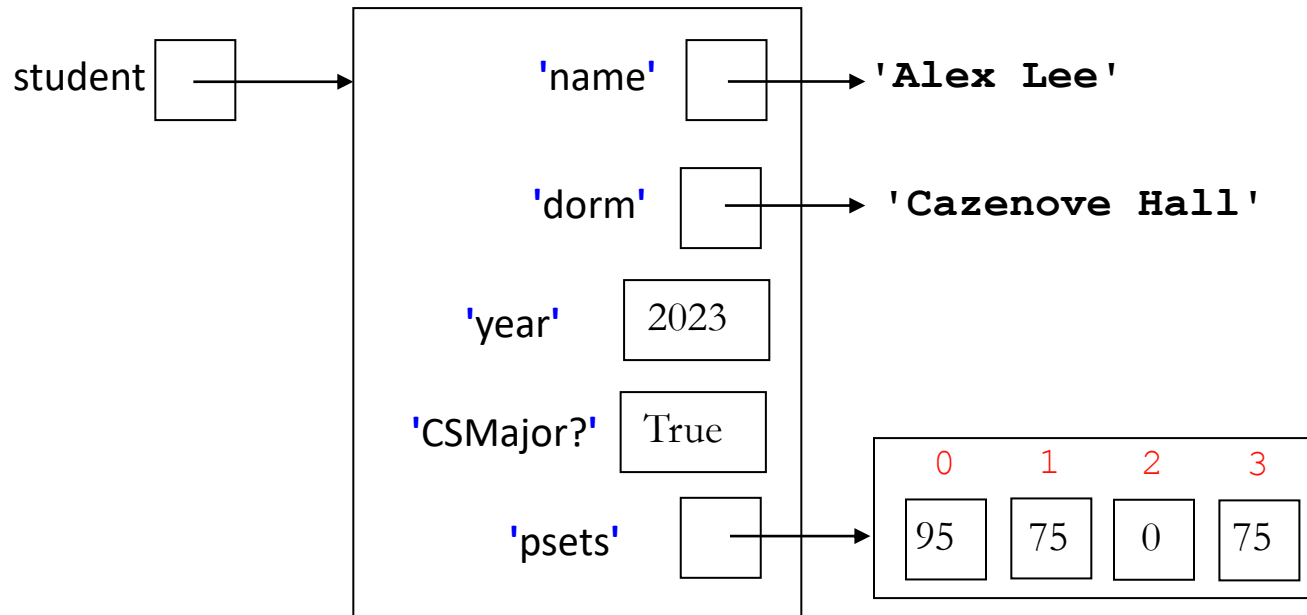
# Dictionaries are Mutable:

## remove key/value pair

### Concepts in this slide:

The method **pop** with dicts always requires an argument. Why?

A *key/value* pair can be removed by the **pop** method, which returns the old value in addition to removing the *key/value* pair:



```
In [20]: student.pop('section')
```

```
Out[20]: 2 # returns the value associated with 'section'
```

```
In [21]: student
```

```
Out[21]: {'CSMajor?': True, 'dorm': 'Cazenove Hall',  
'name': 'Alex Lee', 'psets': [95, 75, 0, 75], 'year':  
2023}
```

# Dictionaries are Mutable: update

## Concepts in this slide:

We can use **update** to change several key/value pairs at one time.

An existing dictionary can be updated with new key/pair values through the **update** method. Here is an example with the `contributions` dictionary:

```
In [22]: contributions
```

```
Out[22]: {'uma52': {2015: 10, 2016: 15},  
          'setam$3': {2012: 23, 2013: 34, 2014: 17},  
          'rid12': {2009: 5, 2010: 18, 2012: 4}}
```

```
In [23]: newDonors = {'max**': {2011: 39, 2013: 27, 2015: 41},  
                     'dev11': {2020: 21}}
```

```
In [24]: contributions.update(newDonors) # no output
```

```
In [25]: contributions
```

```
Out[25]: {'uma52': {2015: 10, 2016: 15},  
          'setam$3': {2012: 23, 2013: 34, 2014: 17},  
          'rid12': {2009: 5, 2010: 18, 2012: 4},  
          'max**': {2011: 39, 2013: 27, 2015: 41},  
          'dev11': {2020: 21}}
```

# Dictionary Methods: **get**

The **get** method is an alternative to using subscripting to get the value associated with a key in a dictionary. It takes two arguments:

- (1) the key
- (2) a default value to use if the key is not in the dictionary

```
In [26]: daysInMonth.get('Oct', 'unknown')  
Out[26]: 31
```

```
In [27]: daysInMonth.get('OCT', 'unknown')  
Out[27]: 'unknown'
```

It is possible to use **get** without a second argument (it is optional). In this case, if the key doesn't exist, **get** will return `None`. To see it, we need to print the value:

```
In [28]: print(daysInMonth.get('OCT'))  
None
```



# Dictionary Methods: keys, values, items

## Concepts in this slide:

All three dict methods: keys, values, items return a list with synchronized order.

The **keys**, **values**, and **items** method invocations on a dictionary return, respectively, objects holding the keys, values, and key/value pairs for a dictionary:

```
In [29]: daysInMonth.keys()
```

```
Out[29]: dict_keys(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',  
'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
```

```
In [30]: daysInMonth.values()
```

```
Out[30]: dict_values([31, 28, 31, 30, 31, 30, 31, 31, 30, 30, 30, 31])  
# Values have same order as keys from .keys() method invocation  
# The type here is dict_values
```

```
In [31]: daysInMonth.items()
```

```
Out[31]: dict_items([('Jan', 31), ('Feb', 28), ('Mar', 31), ('Apr',  
30), ('May', 31), ('Jun', 30), ('Jul', 31), ('Aug', 31), ('Sep', 30),  
('Oct', 30), ('Nov', 30), ('Dec', 31)])  
# Items have same order as other two methods. The type is dict_items.
```

The objects returned by these three methods are so-called **dictionary view objects**. If the underlying dictionary changes after these are created, these dictionary view objects will reflect those changes.

# Iterating over keys in a dictionary

## Concepts in this slide:

Use `for key in dict:` to iterate over the keys of a dictionary; don't use `.keys()`

To iterate over the keys in a dictionary, use  
There is **no need** to use `dict.keys()`!

```
for key in dict :
```

```
In [32]: for num in phones:
          print(phones[num] , num)
```

```
Gal Gadot 5558671234
Trevor Noah 9996541212
Paula A. Johnson 7811234567
```

```
In [33]: for month in daysInMonth:
          print(month, daysInMonth[month])
```

```
May 31
Aug 31
Nov 30
...
```

# Iterating over keys in a dictionary

```
In [33]: for first, last in heroes:
          print(f"{first} {last} is " + \
                f"{heroes[(first, last)][1]} 's alter ego.")
```

These braces are placeholders  
in the format string;  
they have **nothing** to do  
with dictionaries.

```
Diana Prince is Wonderwoman's alter ego.
Peter Parker is Spiderman's alter ego.
Clark Kent is Superman's alter ego.
```

```
In [34]: for month in daysInMonth:
          print(f"*{month} has {daysInMonth[month]} days.*")
```

```
*May has 31 days.*
*Aug has 31 days.*
*Nov has 30 days.*
...
```

# Iterating over/membership in dictionaries



**Digging  
Deeper**

When iterating over the keys in a dictionary, just write

```
for someKey in someDict:
```



rather than

```
for someKey in someDict.keys():
```



because they have a similar meaning, but the latter creates an unnecessary object.

Similarly, when testing if a key is in a dictionary, just write

```
if someKey in someDict:
```



rather than

```
if someKey in someDict.keys():
```



In Python 3, the unnecessary `.keys()` returns a `dict_keys` object that still allows efficient membership tests and iteration, so there's not a big downside to using `.keys()`. But in Python 2, `.keys()` returns a newly constructed list that can lead to significant inefficiencies for big dictionaries.

# Iterating over values & items in a dictionary

**Concepts in this slide:**  
Iterating over the values and key:value pairs in a dictionary.

`.values()` and `.items()` are useful for iterating over values of key:value pairs of dictionary:

```
In [28]: for name in phones.values():  
         print(f"Call {name}!")
```

Call Gal Gadot!

Call Trevor Noah!

Call Paula A. Johnson!



```
In [29]: for number, name in phones.items():  
         print(f"Call {name} at {number}.")
```

Call Gal Gadot at 5558671234.

Call Trevor Noah at 9996541212.

Call Paula A. Johnson at 7811234567.

**To notice:** 

The method `.items` returns a list of tuples, so we can use tuple assignment to assign to the key and value at the same time.

# Summary of dictionary methods

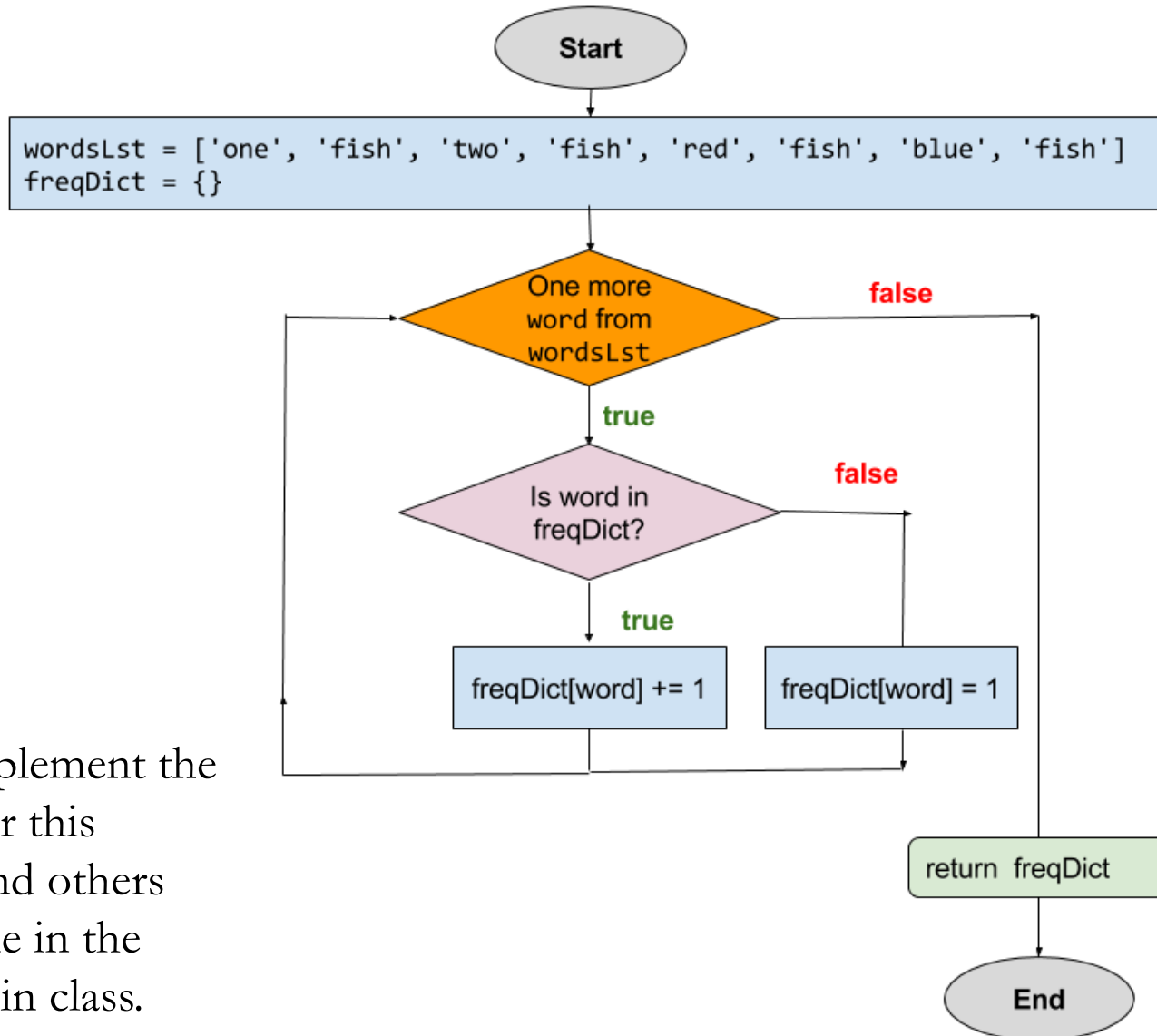
Method	Result	Mutates dict?
<code>.keys()</code>	Returns all keys as a <code>dict_keys</code> object	No
<code>.values()</code>	Returns all values as a <code>dict_values</code> object	No
<code>.items()</code>	Returns (key, value) pairs as a <code>dict_items</code> object	No
<code>.get(key [, val])</code>	Returns corresponding value if <code>key</code> in dict, else returns <code>val</code> . The notation <code>[, val]</code> means that the second argument <code>val</code> is optional and can be omitted. If it is not specified, it defaults to <code>None</code> .	No
<code>.pop(key)</code>	Removes <code>key:val</code> pair with given <code>key</code> from dict and returns associated <code>val</code> . Signals <code>KeyError</code> if <code>key</code> not in dict.	Yes
<code>.update(dict2)</code>	Adds new <code>key:value</code> pairs from <code>dict2</code> to dict, replacing any <code>key:value</code> pairs with existing <code>key</code> .	Yes
<code>.clear()</code>	Removes all items from the dict.	Yes



# An Application for dictionaries: Word Frequencies

## Concepts in this slide:

An algorithm represented as a flow chart diagram to solve a common problem.



We will implement the solution for this problem and others like this one in the Notebook in class.

# Mutability with hash



When trying to use a mutable value as key for a dictionary, we'll get an error:

```
In [34]: daysOfMonth[['Feb', 2015]] = 28
```

```
TypeError: unhashable type: 'list'
```

What does this error mean? It turns out, Python stores keys of a dictionary as hash values, generated by the **hash** function. This is why dictionaries are also known as hashtables, especially in other programming languages.

```
In [31]: hash("Wellesley")
```

```
Out[35]: 1371402960993349759
```

```
In [31]: hash((2015,10))
```

```
Out[36]: 3711745792089893406
```

```
In [31]: hash(1234)
```

```
Out[37]: 1234
```

Only immutable objects have hash values. We'll get an error for mutable objects.

```
In [31]: hash([1,2,3])
```

```
TypeError: unhashable type: 'list'
```

# Summary

1. A dictionary is a new Python data type that is a kind of collection. It differs from lists because it stores together pairs of keys and values. We use keys to access values.
2. Keys are always immutable (numbers, strings, ranges, and tuples), while values can be any Python object.
3. There are at least three different ways to create a dictionary, but the most common one is to start with an empty dict and add keys and values while iterating over some other data sequence.
4. Dictionaries are mutable, we can change the values through their keys, add new key/value pairs, and remove existing ones. Three examples of methods that mutate the dictionary are `.pop`, `.update`, and `.clear`.
5. An important method that avoids encountering the `KeyError` (in case the key doesn't exist) is `.get`, which can be used with one or two arguments.
6. The methods `.keys`, `.values`, and `.items` return dictionary view objects that track later changes to the dictionary.
7. Rather than writing `key in myDict.keys()`, just write `key in myDict` when iterating or testing membership in a dictionary.

# Test your knowledge

1. What is the main difference between data types that are sequences and those that are collections?
2. Would you need to use **range** to generate indices to access the elements of a dictionary? Explain.
3. Which has to be unique: the keys or the values of the dictionary?
4. When iterating over the values of a dictionary as in slide 26, is it possible to access the keys too? Explain. Which of the dictionaries defined in slides 6 and 8 would be a good example to make your point.
5. The diagram in slide 26 shows two boxes for assigning values to the **freqDict[word]**. How can you replace the **if** statement and those two assignments by one of the dictionary methods you learned?