Chapter 10
Dictionaries

A. Downey, *Think Python: How to Think Like a Computer Science,* 3rd ed., O'Reilly, 2024.

https://allendowney.github.io/ThinkPython/

Contents

1. A Dictionary Is a Mapping

- A dictionary is like a list, but more general.
  - In a list, the indices have to be integers; in a dictionary they can be (almost) any type.
  - For example, suppose we make a list of number words, like this:

```
lst = ['zero', 'one', 'two']
lst[1]
```

```
'one'
```

  - But suppose we want to go in the other direction, and look up a word to get the corresponding integer. We can't do that with a list, but we can with a dictionary.

1. A Dictionary Is a Mapping

```
0   -->   'zero'
1   -->   'one'
2   -->   'two'
```

```
'zero'   -->   0
'one'    -->   1
'two'    -->   2
```

```
lst = ['zero', 'one', 'two']
lst[1]
```

```
'one'
```

We can’t do that with a list, but we can with a dictionary.

1. A Dictionary Is a Mapping

- A dictionary contains items which represents the association of a **key** and a **value**. (key: value)
- Creating an empty dictionary {} and assigning it to **numbers**:

```
numbers = {}

print(numbers)
type(numbers)
```

```
{}
dict
```

1. A Dictionary Is a Mapping

- To add items to the dictionary, we'll use square brackets:

```
1 numbers['zero'] = 0
2 numbers
```

```
{'zero': 0}
```

  - This assignment adds to the dictionary an item, which represents the association of a **key** and a **value**.
  - In this example, the key is the string 'zero' and the value is the integer 0.
  - If we display the dictionary, we see that it contains one item, which contains a key and a value separated by a colon.

1. A Dictionary Is a Mapping

- We can add more items like this:

```
numbers['zero'] = 0
numbers['one'] = 1
numbers['two'] = 2
numbers
```

```
{'zero': 0, 'one': 1, 'two': 2}
```

```
numbers['two']
```

```
2
```

```
numbers['three']

```

```
---------------------------------------
KeyError
Cell In[11], line 1
----> 1 numbers['three']

KeyError: 'three'
```

- To look up a key and get the corresponding value, we use the bracket operator.
  - If the key isn't in the dictionary, we get a **KeyError**

1. A Dictionary Is a Mapping

- We can add more items like this:

```
numbers['zero'] = 0
numbers['one'] = 1
numbers['two'] = 2
numbers
```

```
{'zero': 0, 'one': 1, 'two': 2}
```

- The **len** function works on dictionaries; it returns the number of items:

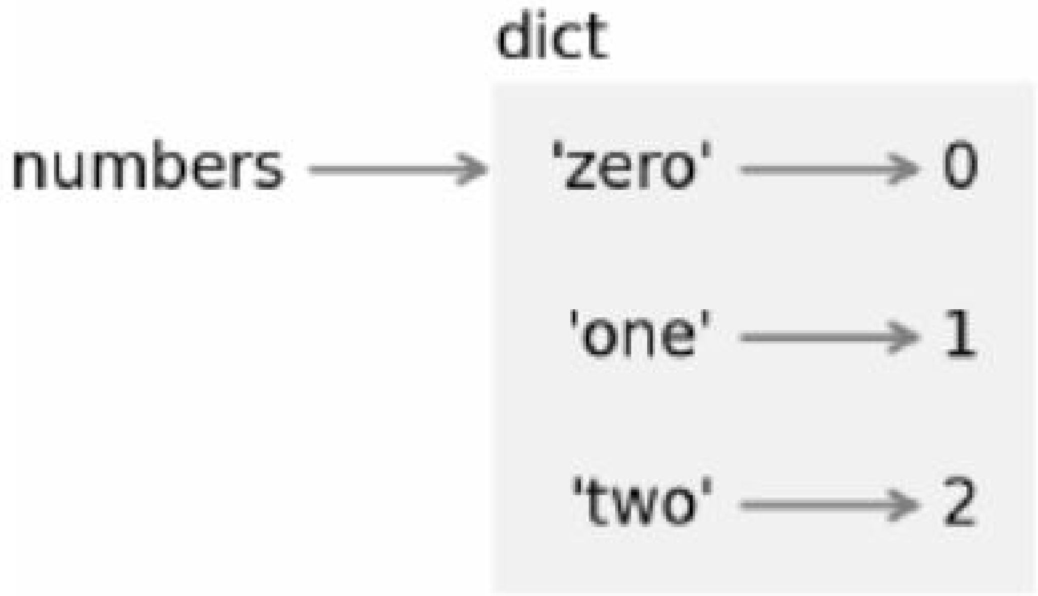
```
numbers['zero'] = 0
numbers['one'] = 1
numbers['two'] = 2

len(numbers)
```

```
3
```

1. A Dictionary Is a Mapping

- In mathematical language, a dictionary represents a mapping from keys to values, so you can also say that each key "maps to" a value.
- In this example, each number word maps to the corresponding integer.



1. A Dictionary Is a Mapping

- We can update the value of a key in a dictionary:

```python
numbers = {}
numbers['one'] = 1
numbers['two'] = 2
print(numbers)

numbers['one'] = 1111
print(numbers)
```

```
{'one': 1, 'two': 2}
{'one': 1111, 'two': 2}
```

2. Creating Dictionaries

- We can create the dictionary all at once:
  - Each item consists of a key and a value separated by a colon. The items are separated by commas and enclosed in curly braces.

```
numbers = {'zero': 0, 'one': 1, 'two': 2}
```

- Another way to create a dictionary is to use the **dict** function. We can make a empty dictionary like this:

```
numbers = dict()
numbers
```

```
{}
```

2. Creating Dictionaries

- We can make a copy of a dictionary like this:

```
numbers = {'zero': 0, 'one': 1, 'two': 2}

numbers_copy = dict(numbers)
numbers_copy
```

```
{'zero': 0, 'one': 1, 'two': 2}
```

3. The **in** Operator

- The **in** operator works on dictionaries, too; it tells you whether something appears as a ***key*** in the dictionary:

```
numbers = {'zero': 0, 'one': 1, 'two': 2}

'one' in numbers
```

```
True
```

- The **in** operator does not check whether something appears as a *value*:

```
numbers = {'zero': 0, 'one': 1, 'two': 2}

1 in numbers
```

```
False
```

3. The **in** Operator

- To see whether something appears as a *value* in a dictionary, you can use the method **values**, which returns a sequence of values, and then use the **in** operator:

```
numbers = {'zero': 0, 'one': 1, 'two': 2}

1 in numbers.values()
```

```
True
```

3. The **in** Operator

- The items in a Python dictionary are stored in a **hash table**, which is a way of organizing data that has a remarkable property: *the in operator takes about the same amount of time no matter how many items are in the dictionary*. That makes it possible to write some remarkably efficient algorithms.
- Write a program to count the number of words in the *words.txt* that their reversed words are in the *words.txt*.

3. The **in** Operator

- Count the number of words in the *words.txt* that their reversed words are in the *words.txt*.

```python
def reverse_word(word):
    return ''.join(reversed(word))

def too_slow(w_list):
    count = 0
    for word in w_list:
        if reverse_word(word) in w_list:
            count += 1
    return count
```

```python
word_list = open('words.txt').read().split()
print(len(word_list))

total = too_slow(word_list)
print(total)
```

```
113783
885
```

```python
def reverse_word(word):
    return ''.join(reversed(word))

def much_faster(w_dict):
    count = 0
    for word in w_dict:
        if reverse_word(word) in w_dict:
            count += 1
    return count
```

```python
word_list = open('words.txt').read().split()
print(len(word_list))

word_dict = {}
for word in word_list:
    word_dict[word] = 1

total = much_faster(word_dict)
print(total)
```

```
113783
885
```

3. The **in** Operator

- In general, the time it takes to find an element in a list is proportional to the length of the list.
- The time it takes to find a key in a dictionary is almost constant—regardless of the number of items.

```
d = {'a': 1, 'b': 2}
d['a'] = 3
d
```

```
{'a': 3, 'b': 2}
```

4. A Collection of Counters

- Suppose you are given a string and you want to count how many times each letter appears. A dictionary is a good tool for this job.

```
def value_counts(string):
    counter = {}
    for letter in string:
        if letter not in counter:
            counter[letter] = 1
        else:
            counter[letter] += 1
    return counter
```

```
counter = value_counts('brontosaurus')
counter
```

```
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
```

5. Looping and Dictionaries

- If you use a dictionary in a **for** statement, it traverses the ***keys*** of the dictionary.

```
def value_counts(string):
    counter = {}
    for letter in string:
        if letter not in counter:
            counter[letter] = 1
        else:
            counter[letter] += 1
    return counter
```

```
counter = value_counts('banana')
counter
```

```
{'b': 1, 'a': 3, 'n': 2}
```

```
for key in counter:
    print(key)
```

```
b
a
n
```

5. Looping and Dictionaries

- To print the values of a dictionary, we can use the **values** method of the dictionary:

```
def value_counts(string):
    counter = {}
    for letter in string:
        if letter not in counter:
            counter[letter] = 1
        else:
            counter[letter] += 1
    return counter
```

```
counter = value_counts('banana')
counter
```

```
{'b': 1, 'a': 3, 'n': 2}
```

```
for value in counter.values():
    print(value)
```

```
1
3
2
```

5. Looping and Dictionaries

- To print the keys and values, we can loop through the keys and look up the corresponding values:

```python
def value_counts(string):
    counter = {}
    for letter in string:
        if letter not in counter:
            counter[letter] = 1
        else:
            counter[letter] += 1
    return counter
```

```python
counter = value_counts('banana')
counter
```

```
{'b': 1, 'a': 3, 'n': 2}
```

```python
for key in counter:
    value = counter[key]
    print(key, ':', value)
```

```
b : 1
a : 3
n : 2
```

6. Lists and Dictionaries

- You can put a list in a dictionary as a value.

```
dict = {4: ['r', 'o', 'u', 's'], "five": 5}
```

```
['r', 'o', 'u', 's']
```

```
dict[4]
```

```
['r', 'o', 'u', 's']
```

```
dict['five']
```

```
5
```

6. Lists and Dictionaries

- But you can’t put a list in a dictionary as a key. Here’s what happens if we try:

```
dict = {}

letters = list('abcd')
dict[letters] = 4
```

```
---------------------------------------------------------------------------
TypeError
Cell In[12], line 4
      1 dict = {}
      3 letters = list('abcd')
----> 4 dict[letters] = 4

TypeError: unhashable type: 'list'
```

6. Lists and Dictionaries

- Dictionaries use **hash tables**, and that means that the keys have to be **hashable**.
- A **hash** is a function that takes a value (of any kind) and returns an ***integer***. Dictionaries use these integers, called **hash values**, to store and look up keys.
- Dictionaries only works if a key is **immutable**, so its hash value is always the same.
- But if a key is mutable, its hash value could change, and the dictionary would not work.
- That's why keys have to be hashable, and why mutable types like lists aren't.
- Since dictionaries are mutable, they can't be used as keys either. But they can be used as values.

7. Accumulating a List

- For many programming tasks, it is useful to loop through one list or dictionary while building another.
  - As an example, we’ll loop through the words in word_dict and make a list of palindromes—that is, words that are spelled the same backward and forward, like “noon” and “rotator.”

```
def reverse_word(word):
    return ''.join(reversed(word))

def is_palindrome(word):
    """Check if a word is a palindrome."""
    return reverse_word(word) == word
```

```
is_palindrome('rotator')
```

```
True
```

7. Accumulating a List

- We can count the number of palindromes in a **word_dict** like this:
  - The pattern is familiar.
    - Before the loop, **count** is initialized to **0**.
    - Inside the loop, if word is a palindrome, we increment **count**.
    - When the loop ends, **count** contains the total number of palindrome

```
word_list = open('words.txt').read().split()
```

```
word_dict = {}
for word in word_list:
    word_dict[word] = 1
```

```
count = 0

for word in word_dict:
    if is_palindrome(word):
        count +=1

count
```

```
91
```

7. Accumulating a List

- We can use a similar pattern to make a list of palindromes:
  - The pattern is familiar.
    - Before the loop, **palindromes** is initialized with an empty list.
    - Inside the loop, if **word** is a palindrome, we append it to the end of **palindromes**.
    - When the loop ends, **palindromes** is a list of palindromes.
  - In this loop, **palindromes** is used as an accumulator, which is a variable that collects or accumulates data during a computation.

```
word_list = open('words.txt').read().split()
```

```
word_dict = {}
for word in word_list:
    word_dict[word] = 1
```

```
palindromes = []

for word in word_dict:
    if is_palindrome(word):
        palindromes.append(word)

palindromes[:5]
```

```
['aa', 'aba', 'aga', 'aha', 'ala']
```

7. Accumulating a List

- Select only palindromes with seven or more letters

```python
def is_palindrome(word):
    """Check if a word is a palindrome."""
    return reverse_word(word) == word
```

```python
word_list = open('words.txt').read().split()

word_dict = {}
for word in word_list:
    word_dict[word] = 1

palindromes = []
for word in word_dict:
    if is_palindrome(word):
        palindromes.append(word)

long_palindromes = []
for word in palindromes:
    if len(word) >= 7:
        long_palindromes.append(word)

long_palindromes

```

```
['deified', 'halalah', 'reifier', 'repaper', 'reviver', 'rotator', 'sememes']
```

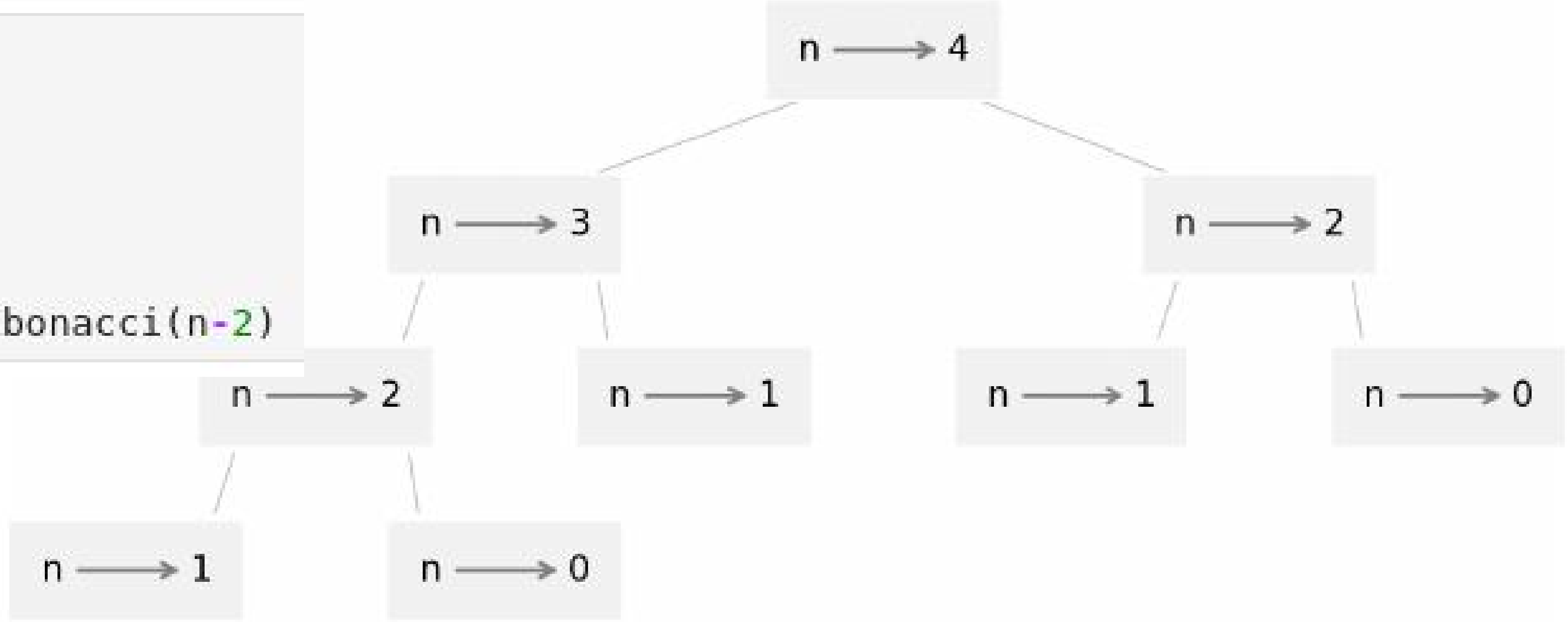
Looping through a list like this, selecting some elements and omitting others, is called **filtering**.

8. Memos

- If you ran the recursive fibonacci function from, maybe you noticed that the bigger the argument you provide, the longer the function takes to run:

```python
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

The **call graph** for **fibonacci** with $n = 4$:



How many times **fibonacci(0)** and **fibonacci(1)** are called. This is an inefficient solution to the problem, and it gets worse as the argument gets bigger.

8. Memos

- One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **memo**. Here is a "memoized" version of **fibonacci**:

```python
known = {0:0, 1:1}

def fibonacci_memo(n):
    if n in known:
        return known[n]
    res = fibonacci_memo(n-1) + fibonacci_memo(n-2)
    known[n] = res
    return res
```

```python
fibonacci_memo(40)
```

```
102334155
```

8. Memos

- Comparing the two functions **fibonacci(40)** takes about 30 seconds to run. **fibonacci_memo(40)** takes about 30 miliseconds.