PROGRAMMING FUNDAMENTALS

DATA TYPES: DICTIONARIES

João Correia Lopes

INESC TEC, FEUP

15 November 2018

GOALS

By the end of this class, the student should be able to:

- Use the main operation and methods available to work with dictionaries
- Describe the differences between dictionaries aliasing and shallow copying

BIBLIOGRAPHY

- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3, 2018 (Section 5.4) [PDF]
- Brad Miller and David Ranum, Learning with Python: Interactive Edition. Based on material by Jeffrey Elkner, Allen B. Downey, and Chris Meyers (Chapter 120) [HTML]
- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3 (RLE), 2012 (Chapter 20) [HTML]

TIPS

- There’s no slides: we use a script, illustrations and code in the class. Note that this PDF is NOT a replacement for **studying the bibliography** listed in the *class plan*
- “Students are responsible for anything that transpires during a class—therefore **if you’re not in a class**, you should get notes from someone else (not the instructor)”—David Mayer
- The best thing to do is to **read carefully** and **understand** the documentation published in the Content wiki (or else **ask** in the recitation class)
- We will be using **Moodle** as the primary means of communication

CONTENTS

1 DATA TYPES: DICTIONARIES
- 5.1.1 A compound data type
- 5.4.1 Dictionary operations
- 5.4.2 Dictionary methods
- 5.4.3 Aliasing and copying
- 5.4.4 Counting letters
- 20.4 Sparse matrices
- 20.5 Memoization

A COMPOUND DATA TYPE

- So far we have seen built-in types like `int`, `float`, `bool`, `str` and we've seen lists, pairs or tuples
- Strings, lists, and tuples are qualitatively different from the others because they are made up of smaller pieces
- Lists, tuples, and strings have been called *sequences*, because their items occur in order
- Dictionaries group any number of items, of different types, into a single compound value
- Dictionaries are not sequences!

DICTIONARY

- Dictionaries are yet another kind of compound type
- They are Python’s built-in **mapping type**
- They map **keys**, which can be any immutable type, to **values**, which can be any type (heterogeneous) [1]
- In other languages, they are called *associative arrays* since they associate a key with a value
- One way to create a dictionary is to start with the empty dictionary and add **key:value** pairs

```
    >>> english_spanish = {}
    >>> english_spanish[’one’] = "uno"
    >>> english_spanish["two"] = ’dos’
    >>> print(english_spanish)
    {’one’: ’uno’, ’two’: ’dos’}
```

[1] Just like the elements of a list or tuple

Hashing

- The order of the pairs may not be what was expected
- Python uses complex algorithms, designed for very fast access, to determine where the key:value pairs are stored in a dictionary
- For our purposes we can think of this ordering as **unpredictable**
- The implementation uses a technique called **hashing**
- The same concept of mapping a key to a value could be implemented using a list of tuples, but…

```
>>> {"apples": 430, "bananas": 312, "oranges": 525, "pears": 217}
{'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

>>> [("apples", 430), ("bananas", 312), ("oranges", 525), ("pears", 217)]
[('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]
```

LOOK UP A VALUE

DICIO-NÁRIO
DESSE JEITO VOCÊ NUNCA VAI TERMINAR DE LER UM LIVRO TÃO GROSSO!

⇒ https://en.wikipedia.org/wiki/Mafalda

LOOK UP A VALUE

- Another way to create a dictionary is to provide a list of **key:value** pairs using the same syntax as the previous output
- It doesn't matter what order we write the pairs (there's no indexing!)[2]

```
>>> english_spanish = {"one": "uno", "three": "tres", "two": "dos"
    }
>>> english_spanish
{'one': 'uno', 'three': 'tres', 'two': 'dos'}

>>> print(english_spanish["two"])
dos
```

[2] The dictionary is the first compound type that we've seen that is not a sequence, so we can't index or slice a dictionary

DICTIONARY OPERATIONS

- The `del` statement removes a *key:value* pair from a dictionary
- The len function also works on dictionaries; it returns the number of *key:value* pairs

```
>>> inventory = {"apples": 430, "bananas": 312, "oranges": 525, "pears": 217}
>>> del inventory["bananas"]
>>> len(inventory)
```

⇒ https://github.com/fpro-admin/lectures/blob/master/14/operations.py

DICTIONARY METHODS

- Dictionaries have a number of useful built-in methods
- The `keys` method returns what Python3 calls a `view` of its underlying keys
    - A view object has some similarities to the range object we saw earlier — it is a lazy promise, to deliver its elements when they're needed by the rest of the program
    - We can iterate over the view, or turn the view into a list like this
- The `values` method is similar
- The `items` method also returns a view, which promises a list of tuples

```
for key in english_spanish.keys():  # The order of the k's is not defined
    print("Got key", key, "which maps to value", english_spanish[key])
```

⇒ https://github.com/fpro-admin/lectures/blob/master/14/methods.py

ALIASING AND COPYING

- As in the case of lists, because **dictionaries are mutable**, we need to be aware of aliasing
- Whenever two variables refer to the same object, changes to one affect the other
- If we want to modify a dictionary and keep a copy of the original, use the `copy` method

```
>>> opposites = {"up": "down", "right": "wrong", "yes": "no"}
>>> alias = opposites
>>> copy = opposites.copy()  # Shallow copy
```

⇒ https://github.com/fpro-admin/lectures/blob/master/14/methods.py

GENERATE A FREQUENCY TABLE

- To write a function that counted the number of occurrences of a letter in a string
- Dictionaries provide an elegant way to generate a frequency table

```
      start with an empty dictionary
      for each letter in the string:
         find the current count (possibly zero) and increment it
      the dictionary contains pairs of letters and their frequencies
```

⇒ https://github.com/fpro-admin/lectures/blob/master/14/frequency-table.py

SPARSE MATRICES

- We previously used a list of lists to represent a matrix
- That is a good choice for a matrix with mostly nonzero values, but consider a sparse matrix like this one:

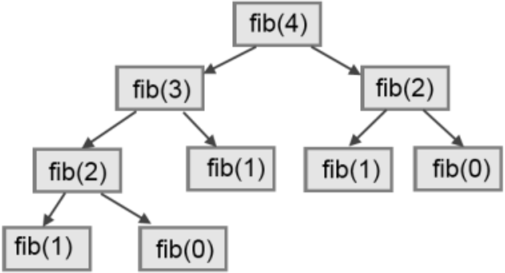
$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

- The list representation contains a lot of zeroes
- An alternative is to use a dictionary and the `get()` method

⇒ https://github.com/fpro-admin/lectures/blob/master/14/matrix.py

MEMOIZATION

- Consider this call graph for `fib()` with n = 4
- A good 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**



⇒ https://github.com/fpro-admin/lectures/blob/master/14/fib.py

EXERCISES

- Moodle activity at: LE14: Dictionaries