UNIVERSITY OF TWENTE.

Dictionaries

Their data structure, and how to work with them

1

Contents

- Dictionary syntax
- Building a dictionary
- Accessing data inisde a dictionary
- Dictionary methods
- Dictionary as a matrix

UNIVERSITY OF TWENTE.

2

Built-in Python data structures

Python knows a number of built-in *compound* data types (containers), used to group objects.

Sequences

- Types: strings, lists, tuples
- Operations: Indexing, slicing, adding, multiplying, iteration & membership

Dictionaries

- Map keys to values through index
- Suitable for unstructured data

Sets

- Unordered and do not map keys to values

Dictionaries

- A dictionary is a *mapping* data structure
- It establishes a relationship between a **key** and a **value,** and maps each key to some value
- For instance, think about *capitals* as a dictionary, which uses a country name as key. *Capitals* would map *'Belgium'* to *'Brussels'*.

- Handy to store data organized by name, rather than position:
  - Index of a programming book
  - Contacts in your phone
  - Temperature records for each city of a country

Basics: Creating a dictionary

Dictionaries can be defined in two ways:

- Notation that uses curly brackets: ***{*** and ***}***
- Notation that uses the built-in function ***dict()***

```
landcover = {}                           # the empty dictionary
landcover["Natural grasses"] = 45        # this adds one entry to the dictionary
landcover["Agricultural grasses"] = 1
landcover["Deciduous"] = 11
landcover["Coniferous"] = 12
```

UNIVERSITY OF TWENTE.

5

Basics: Creating a dictionary

Now, we can take a look at its structure by printing the whole dictionary

```
print(landcover)
{'Natural grasses': 45, 'Agricultural grasses': 1, 'Deciduous': 11, 'Coniferous': 12}
```

Important

- Dictionaries are **not** ordered data structures
- Dictionaries are **not** sequences

UNIVERSITY OF TWENTE.

6

Basics: Creating a dictionary

We can also do it in one go:

landcover = {"Natural grasses":45, "Deciduous":11, "Coniferous":12, "Agricultural grasses":1}

An element is accessed using its key:

landcover["Coniferous"]

12

landcover["Natural grasses"]

45

UNIVERSITY OF TWENTE.

7

Basics: Keys and values

Dictionaries store **key:value** pairs

Keys must be immutable:

- Types of key:
  - Number : *landcover[45] = "Natural grasses"*
  - String : *landcover["Natural grasses"] = 45*
  - Tuple : *landcover[("Enschede", "52N","3E")] = 45*

  - But a key cannot be a list:

  landcover[["Enschede", latitude, longitude]] = 45

  TypeError: unhashable type: 'list'

UNIVERSITY OF TWENTE.

8

Basics: Keys and values

Values can be anything!

- If Enschede municipality has multiple land covers, we can store them all:

landcover["Enschede"] = ("coniferous", "built-up","agricultural grasses", "orchard", "reed")

Here as a tuple; could also have been a list.

UNIVERSITY OF TWENTE.

9

Basics: Keys and values

We can build our dictionary using the built-in **dict()** function:

mixed_forest = ("coniferous", "deciduous")
landcover = dict()
landcover["Hengelo"] = mixed_forest
landcover["Amsterdam"] = ("Built-up", "Greenhouses")

UNIVERSITY OF TWENTE.

10

Basics: Keys and values

With dict() function, we can define the dictionary at once, too:

mixed_forest = ("coniferous", "deciduous")
urban_area = ("built-up", "greenhouses")
landcover = dict(Veluwe = mixed_forest, Amsterdam = urban_area)

UNIVERSITY OF TWENTE.

11

Dictionary methods

Similar to lists and strings, dictionary methods are called like:

object.method()

Important methods:

- *.keys()*: returns a list of all the keys
- *.values()*: returns a list of the values
- *.items()*: returns the (key,value) pairs as a list

View: [(key, value), (key, value) … (key, value)]

UNIVERSITY OF TWENTE.

12

Updating a dictionary

Changing values:

landcover["agriculture"] = 999

landcover["grasses"] = 13

print(landcover)

{ 'agriculture': 999, 'coniferous': 12, 'deciduous': 11, 'grasses': 13 }

UNIVERSITY OF TWENTE.

14

Updating a dictionary

landcover = {'agriculture': 1, 'coniferous': 12,'deciduous': 11, 'grasses': 45}

Removal of an item with a given key is called "popping the item."

an_item = landcover.pop("grasses")

Populating a dictionary with the items of another dictionary:

new_landcovers = {"reed": 36, "greenhouse": 20}

landcover.update(new_landcovers)

print(landcover)

{'agriculture': 1, 'coniferous': 12, 'deciduous': 11, 'grasses': 45, 'reed':36, 'greenhouse':20}

UNIVERSITY OF TWENTE.

15

Updating dictionaries

landcover = {'agriculture': 1, 'coniferous': 12, 'deciduous': 11, 'grasses': 45}

Deletion of a single item:

del landcover["deciduous"]

print(landcover)

{'agriculture': 1, 'coniferous': 12, 'grasses': 45}

Removal of all items:

landcover.clear()

{}

UNIVERSITY OF TWENTE.

16

Iterations

Dictionary methods return lists, and lists are iterable!

Problem:

> *Imagine that you have in a dictionary the **land covers for each city** in the Netherlands. You want to **print the names** of those cities that have patches of **coniferous** forest. You also want to **count** the number of cities with coniferous **patches** in the country. The dictionary has as **key** the **name** of the city, and as **value** the **tuple** with the associated landcovers.*

What computing strategy do you use to achieve this?

UNIVERSITY OF TWENTE.

17

Iterations

```
for key in landcover.keys():
   if "Coniferous" in landcover[key]:
      print("City: ", key)

patches = 0
for value in landcover.values():
   if "Coniferous" in value:
      patches += 1
print("Cities with coniferous patches: ", patches)
```

Can obviously combine the two computations in a single for loop. Which one?

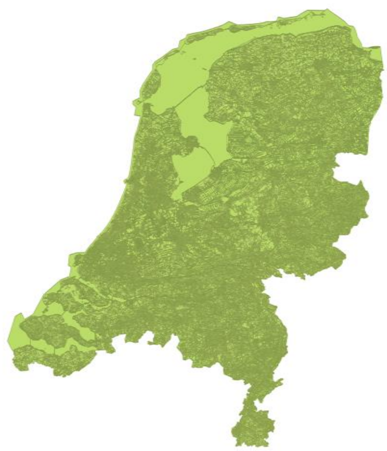
UNIVERSITY OF TWENTE.

18

Vectors or rasters as dictionaries

Vector or raster files can be seen as dense matrices or as dictionaries
Storage of all values in memory may consume substantial resources
Not a wise choice when a significant percentage of the data is not needed
Solution:

- Store only the useful values
- Make use of a sparse matrix



UNIVERSITY OF TWENTE.

19

Rasters or matrices as dictionaries

At 1km pixel resolution:

- NL has 105,000 pixels
- Half of them fall over:
  - Water
  - Belgium or Germany
- Why would one store these values?

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

UNIVERSITY OF TWENTE.

20

Rasters or Matrices as dictionaries

We only need to keep the positions of the non-zero values and write them down

That is:

- Row 0, Col 3, Value 1
- Row 2, Col 1, Value 2
- Row 4, Col 3, Value 3

Which gives:

matrix = {(0, 3):1, (2, 1): 2, (4, 3): 3}

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

UNIVERSITY OF TWENTE.

21

Vectors or rasters as dictionaries

Creation of a sparse matrix from a dense matrix:

```
dense_matrix=[[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]]

sparse_matrix = {}
for i in range(5):
  for j in range(5):
    if dense_matrix[i][j] != 0:
      key = (i, j)
      value = dense_matrix[i][j]
      sparse_matrix[key] = value
print(sparse_matrix)
```

UNIVERSITY OF TWENTE.

22

Summary

- A dictionary is a data structure that stores **key:value** pairs, thus, mapping keys to values
- These pairs are not ordered
- The keys held in a dictionary are immutable, but their associated values are modifiable
- The methods *keys(), values()* and *items()* allow the iteration over and viewing of the dictionary content

UNIVERSITY OF TWENTE.

23

UNIVERSITY OF TWENTE.

Sets

Their data structure and their operations

24

Objectives

After this lecture, students can

- Describe the difference between sets and other data structures, and decide when to use sets in a given problem setting
- Illustrate the syntax of sets
- Code around sets

UNIVERSITY OF TWENTE.

25

25

Contents

- Set syntax
- Set operations
- Set methods
- Difference between set methods and set operations
- Iteration

Python data structures

Sequences
- Types: strings, lists and tuples
- Operations: Indexing, slicing, adding, multiplying, iteration and membership

Dictionaries
- Map keys to values through index
- Suitable for unstructured data

Sets
- Unordered and do not map keys to values

Set definition

Unordered collection of unique and immutable objects

a = set({ 'Landsat 1', 'Landsat 2', 'Landsat 3' })

A set cannot contain object of a mutable type (so no lists and dictionaries) but a set itself is *mutable* (insert or remove element objects), so a set cannot be the member of another set.

However, …

Trick to have set inside another set

To store a set inside another set you need to call the built-in function ***frozenset()*** which will create an immutable set

b = frozenset({ 'Landsat 2', 'Landsat 3' })
c = { 'Landsat 1', b }
print(c)
{ 'Landsat 1', frozenset({ 'Landsat 2', 'Landsat 3' }) }

Set creation

Use the *set()* built-in function and pass a collection of values

```
a = set( 'Landsat 1' )
print(a)
{'d', 'L', 's', 'n', 'a', 't', '1', ' '}

b = set( [ 'Landsat 1', 'Landsat 2', 'Landsat 3' ] )          # convert from list
print(b)
{ 'Landsat 1', 'Landsat 2', 'Landsat 3' }
```

Use curly braces

```
a = { 'Landsat 1', 'Landsat 2', 'Landsat 3' }
b = { 'Landsat 1', 'Landsat 2', 'Landsat 3', 4, 5 }
```

Observe that sets can have members of different type.

Member uniqueness

The elements in a set are unique, therefore, if you create a set with repeated elements, they will count for just one.

```
c = { 'Landsat 1', 'Landsat 1', 'Landsat 2', 'Landsat 3' }
print(c)
{ 'Landsat 1', 'Landsat 2', 'Landsat 3' }


{ 'Landsat 1', 'Landsat 1' } == { 'Landsat 1' }
True
```

You will find out that when working with really large sets, the characteristic of member uniqueness will make us *pay in performance* of operations. For instance, the union of two sets A and B, both with perhaps millions of members, will need to ensure that the result is a again a set, thus with member uniqueness.

Set operations

Sets support operations that we already know from mathematical set theory:

- Set difference
- Set intersection
- Set union
- Symmetric set difference
- Subset test
- Superset test

Operations involving sets

a = { 'Landsat 1', 'Landsat 2' }
b = { 'Landsat 1', 'Landsat 2', 'Landsat 3' }

Difference

a – b
set()
b – a
{ 'Landsat 3' }

Intersection

a & b *# Note the notation*
{ 'Landsat 1', 'Landsat 2' }

Union

a | b *# Note the notation*
{ 'Landsat 1', 'Landsat 2', 'Landsat 3' }

More set operations

a = { 'Landsat 1', 'Landsat 2' }
b = { 'Landsat 1', 'Landsat 2', 'Landsat 3' }

Symmetric difference

a ^ b
{ 'Landsat 3' }

Subset

a <= b
True

Superset

a >= b
False

Set operators as methods

a = { 'Landsat 1', 'Landsat 2' }
b = { 'Landsat 1', 'Landsat 2', 'Landsat 3' }

	Method	Operator	Result
Symmetric difference	a.symmetric_difference(b)	a ^ b	{ 'Landsat 3' }
Subset	a.issubset(b)	a <= b	True
Superset	a.issuperset(b)	a >= b	False
Difference	a.difference(b)	a – b	set()
Intersection	a.intersection(b)	a & b	{ 'Landsat 1', 'Landsat 2' }
Union	a.union(b)	a | b	{ 'Landsat 1', 'Landsat 2', 'Landsat 3' }

Methods versus operations

The methods *union, intersection, difference, symmetric_difference*, *issubset*, and *issuperset* accept any iterable as an argument. However:

{ 'Landsat 1', 'Landsat 2' } | 'Landsat 3'
TypeError: unsupported operand type(s) for |: 'set' and 'str'
{ 'Landsat 1', 'Landsat 2' } | ['Landsat 3']
TypeError: unsupported operand type(s) for |: 'set' and 'list'
{ 'Landsat 1', 'Landsat 2' } | { 'Landsat 3': 1978 }
TypeError: unsupported operand type(s) for |: 'set' and 'dict'

{ 'Landsat 1', 'Landsat 2' }.union('Landsat 3')
{'d', 'L', 's', 'n', 'Landsat 1', 'a', 't', ' ', 'Landsat 2', '3'}
{ 'Landsat 1', 'Landsat 2' }.union(['Landsat 3'])
{'Landsat 1', 'Landsat 2', 'Landsat 3'}
{ 'Landsat 1', 'Landsat 2' }.union({ 'Landsat 3': 1978 })
{'Landsat 1', 'Landsat 2', 'Landsat 3'}

Iteration over a set

a = { 'Landsat 1', 'Landsat 2', 'Landsat 3' }
for item in a:
print (item)
Landsat 2
Landsat 3
Landsat 1

It is like looping over a list, but the order in which items are handled is arbitrary, and may not be the same between different runs of the loop.

Why would one use a set?

Filter duplicates out of some collection

- *Elements in sets are unique*

Isolate differences in lists, strings and other iterable objects

- *Operations corresponding to mathematical set theory*

Perform order-neutral equality

- *Sets are unordered data structures*

Summary

A set is an unordered collection of unique and immutable objects

A set is neither a sequence nor a mapping

Use *set()* or *{…}* to create a new set

Sets support operations corresponding with mathematical set theory; theoretical papers on algorithms often use sets to define the algorithm

Operations and methods are slightly different

Sets support coded iteration

Sets are useful to filter and to compare collections