Fundamentos de Programação

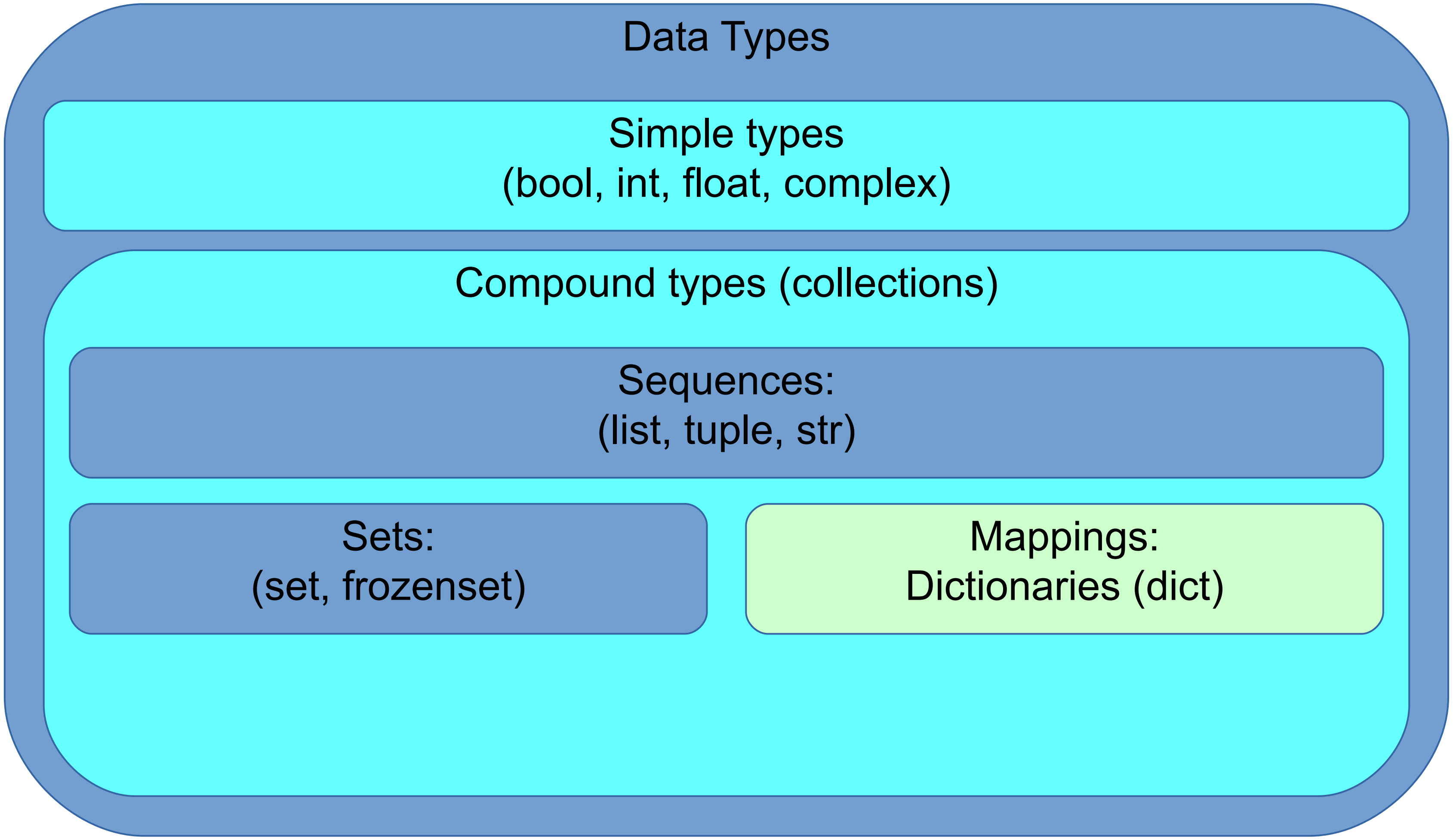
António J. R. Neves
João Rodrigues

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

Resumo

- Dictionaries

Dictionaries



Dictionaries

- A **dictionary** is an *unordered*, *associative collection* of *unique* items.
  - **Collection** because it may contain zero or more items.
  - **Associative** because each item associates a **key** to a **value**.
  - **Unique** because no two items can have the same key.
  - **Unordered** because the order of the items does not matter for equality. *However, items are kept in insertion order (guaranteed since Python 3.7)*.
- Dictionaries are also called **associative arrays** or **maps**.
  - Because they establish a *mapping* between keys and values.
- Dictionary items are also called **key-value pairs**.
- More on dictionaries in the official Python tutorial.

Dictionaries: creating and accessing items

- A dictionary may be created using braces (curly brackets).

```
eng2sp={'one': 'uno', 'two': 'dos', 'three': 'tres'}
shop = {'eggs': 12, 'sugar': 1.0, 'coffee': 3}
```

- An empty dictionary may be created with `{}` or `dict()`.
- To access the value for a given key, use square brackets.

```
shop['sugar']     #-> 1.0
eng2sp['two']     #-> 'dos'
```

Play ▸

- Dictionaries are **mutable**.

```
shop['bread'] = 6   # Add a new key-value association
shop['eggs'] = 24   # Change the value for an existing key
```

Dictionaries: value and key types

- **Values** in a dictionary can be of any type.

```
shop['eggs'] = [1, 'a']
shop['eggs'] = {'brown': 6, 'white': [2, 3]}
```

- **Keys** may be ints, floats, strings, tuples or essentially any other immutable objects. So, lists are not valid keys!

```
eng2sp[4] = 'quatro'       # integer key is fine
d[(12,25)] = 'Christmas'   # tuple key is fine
d[[1,2]] = 'A'             #-> TypeError: unhashable type
```

- Actually, keys must be ***hashable***. In practice, this means:
  - keys must be immutable scalars or
  - immutable collections containing only hashable elements.

```
d[(1,[2,3])] = 'quatro'    #-> TypeError: unhashable type
```

Dictionaries *versus* lists

- When accessing items, a dictionary is a kind of generalized list. In a list, the indices must be integers. In a dictionary, keys can be other kinds of object.

```
lst = [50, 51, 52]
dic = {'um':1, 'vinte':20, 'mil':1000}
lst[1]          #-> 51
dic['mil']      #-> 1000
```

- Unlike lists, the order of items in a dictionary is irrelevant.

```
{'a':1, 'b':2} == {'b':2, 'a':1}    #-> True
[1, 2] == [2, 1]                    #-> False
```

- And you cannot take slices from dictionaries!

```
d = {10:'dez', 20:'vinte', 1000:'mil'}
d[10:20]     # NONSENSE! -> TypeError
```

Dictionary operations

- The `len` function returns the number of key-value pairs.
- The **`in`** operator tells you whether something appears as a *key* in the dictionary. (This is <u>fast</u>!)

```
'two' in eng2sp        #-> True ('two' is a key)
'uno' in eng2sp        #-> False ('uno' is not a key)
```

- Three methods return sequences of keys, values and items.

```
d.keys()    #-> [10, 20, 1000]
d.values()  #-> ['dez', 'vinte', 'mil']
d.items()   #-> [(10, 'dez'), (20, 'vinte'), (1000, 'mil')]
```

- So, to see whether something is a value in the dictionary, you could use (but this is <u>slow</u>):

```
'uno' in eng2sp.values()   #-> True
```

Dictionary methods

- Trying to access an inexistent key is an error.

```
d[10]     #-> 'dez'
d[0]      #-> KeyError
```

Play ▸

- But using the `get` method will return a default value.

```
d.get(10)              #-> 'dez'   (same as d[10])
d.get(0)               #-> None    (no error!)
d.get(0, 'nada')       #-> 'nada'  (no error)
0 in d                 #-> False   (.get did not change d)
print(d)    # {10:'dez', 20:'vinte', 1000:'mil'}
```

- The `setdefault` method is similar, but it also creates a new item if it was missing!

```
d.setdefault(0, 'nada')   #-> 'nada'
0 in d                    #-> True
print(d)    # {10:'dez', 20:'vinte', 1000:'mil' 0:'nada'}
```

Dictionary methods (2)

- Use `pop(key)` to remove the item with the given key and return its value.

```
d = {10:'dez', 20:'vinte', 1000:'mil'}
x = d.pop(10)     #-> x == 'dez'
print(d)          # {20:'vinte', 1000: 'mil'}
```

Play ▸

- We can also delete an item with the **`del`** operator.

```
del d[20]
print(d)          # {1000:'mil'}
```

- The `popitem()` method removes one (unspecified) item from the dictionary, and returns its (key, value) pair.

```
d = {10:'dez', 20:'vinte', 1000:'mil'}
t = d.popitem()   #-> (1000,'mil')
print(d)          # {10:’dez’, 20:'vinte'}
```

Dictionary traversal

- The **`for`** instruction may be used to traverse dictionary *keys*.

```
shop = {'eggs':24, 'bread':6,
        'coffee':3, 'sugar':1.0}
for k in shop:
    print(k, shop[k])
```

Play ▸

```
eggs 24
bread 6
sugar 1.0
coffee 3
```

- This is equivalent to:

```
for k in shop.keys():
    print(k, shop[k])
```

- We may also traverse (key, value) pairs directly:

```
for k, v in shop.items():
    print(k, v)
```

Dictionaries: examples

- Suppose you are given a string and you want to count how many times each letter appears there:

```
message = 'parrot'
d = dict()
for c in message:
    if c not in d:
        d[c] = 1
    else:
        d[c] += 1
```

Play ▸

- To show the results, traverse the keys with a **`for`** statement:

```
for c in d:
    print(c, d[c])
```

Dictionaries: examples (2)

- Create a dictionary that maps from frequencies to letters:

```
inverse = dict()
for key in d:
    val = d[key]
    if val not in inverse:
        inverse[val] = [key]
    else:
        inverse[val].append(key)

print(d)            # from previous slide
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
print(inverse)
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

Dictionaries: updating

- Many algorithms require updating a dictionary one item at a time.
- This can be done in several alternative, but equivalent ways.
- Example: counting characters in a message.

```
#A
d = {}
for c in message:
    if c not in d:
        d[c] = 1
    else:
        d[c] += 1
```

```
#B
d = {}
for c in message:
    if c not in d:
        d[c] = 0
    d[c] += 1
```

```
#C
d = {}
for c in message:
    d[c] = d.get(c, 0) + 1
```

```
#D
d = {}
for c in message:
    d.setdefault(c, 0)
    d[c] += 1
```

Dictionaries: updating (2)

- Example: grouping words in lists according to word length.

```
#A
d = {}
for w in wordlist:
    k = len(w)
    if k not in d:
        d[k] = [w]
    else:
        d[k].append(w)
```

```
#B
d = {}
for w in wordlist:
    k = len(w)
    if k not in d:
        d[k] = []
    d[k].append(w)
```

```
#C
d = {}
for w in wordlist:
    k = len(w)
    d[k] = d.get(k, [])
    d[k].append(w)
```

Play ▸

```
#D
d = {}
for w in wordlist:
    k = len(w)
    d.setdefault(k, []).append(w)
```

```
wordlist=['to','be','or','not','to','be','that','is','the','question']
#A, B, C or D...
d -> {2: ['to', 'be', 'or', 'to', 'be', 'is'], 3: ['not', 'the'],
4: ['that'], 8: ['question']}
```

Dictionaries and lists of tuples

- Method `items` returns a sequence of tuples, where each tuple is a key-value pair.

```
d = {'a':0, 'b':1, 'c':2}
t = d.items() #-> dict_items(('a', 0), ('c', 2), ('b', 1))
```

- We can use a list of tuples to initialize a new dictionary:

```
t = [('a', 0), ('c', 2), ('b', 1)]
d = dict(t)   #-> {'a': 0, 'c': 2, 'b': 1}
```

- Combining items, tuple assignment and for:

```
for key, val in d.items():
    print(val, key)
```

Exercises

- Do these Codecheck exercises.