Fundamentos de Programação

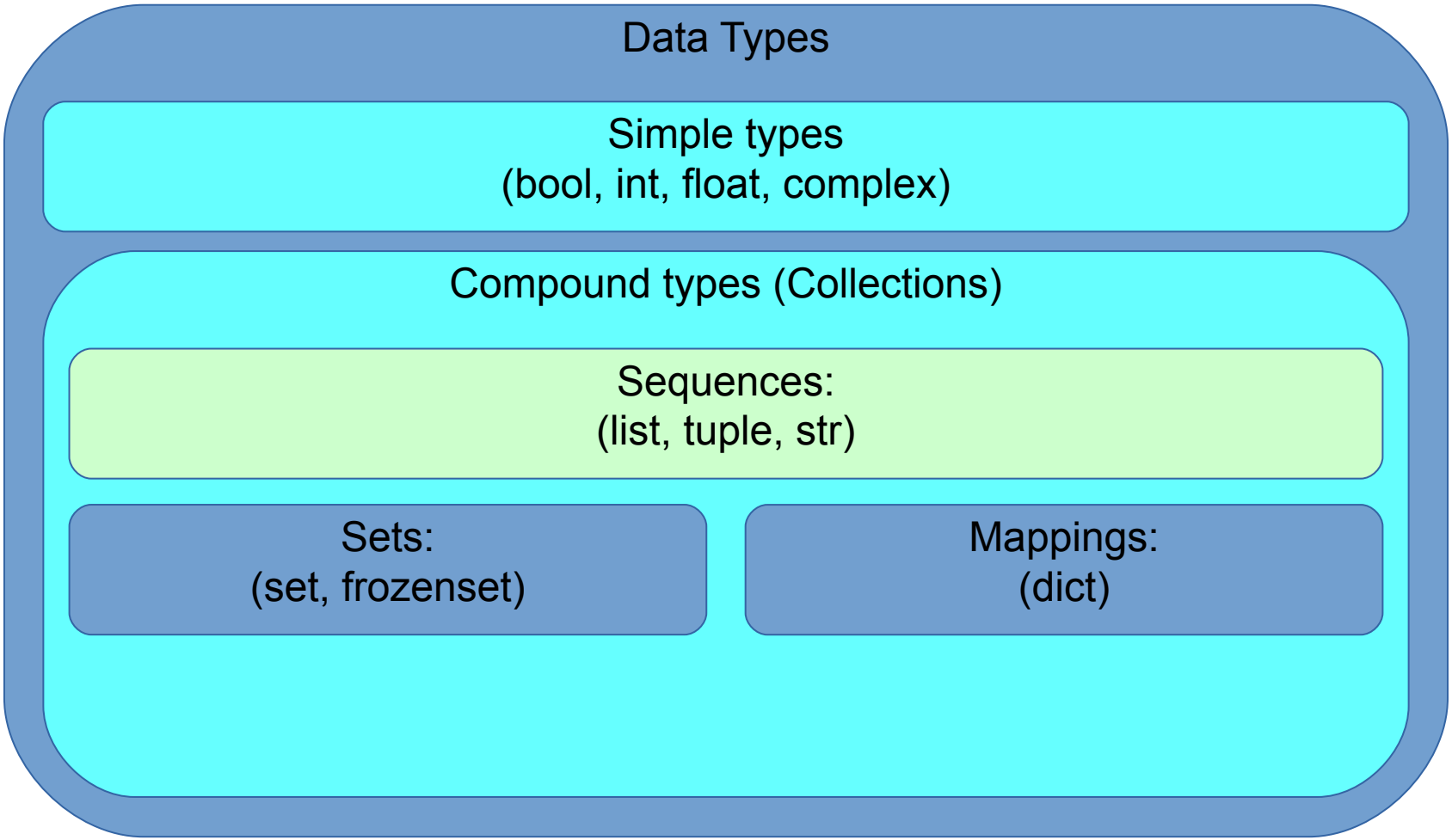
António J. R. Neves
João Rodrigues

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

Summary

- Sequence types
  - Lists
  - Tuples
  - Strings

Sequences


Data Types
Simple types
(bool, int, float, complex)
Compound types (Collections)
Sequences:
(list, tuple, str)
Sets:
(set, frozenset)
Mappings:
(dict)

Lists

- A **list** is a mutable sequence of values of any type.
- The values in a list are called *elements* or *items*.
- List literals are written in brackets.

Play ▸

```
numbers = [10, 20, 30, 40]
fruits = ['banana', 'pear', 'orange']
empty = []                          # an empty list
things = ['spam', 2.0, [1, 2]]  # a list inside a list!
```

- Function `len` returns the *length* of a collection.

```
len(numbers)    #-> 4
len(empty)      #-> 0
len(things)     #-> 3
```

Indexing

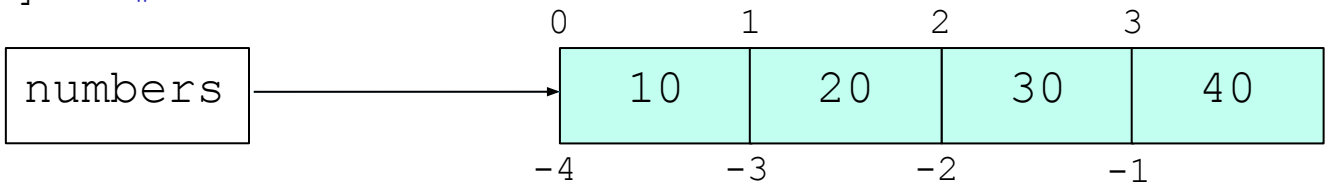
- We can access each element of a sequence using the bracket operator and a value – the *index*.

```
numbers[0]     #-> 10          (index starts at 0)
fruits[2]      #-> 'orange'
```

Play ▸

- A negative index counts backward from the end.

```
numbers[-1]    #-> 40
```



- Any integer expression may be used as an index.

```
numbers[(9+1)%4]   #-> 30
```

- Using an index outside the list bounds is an error.

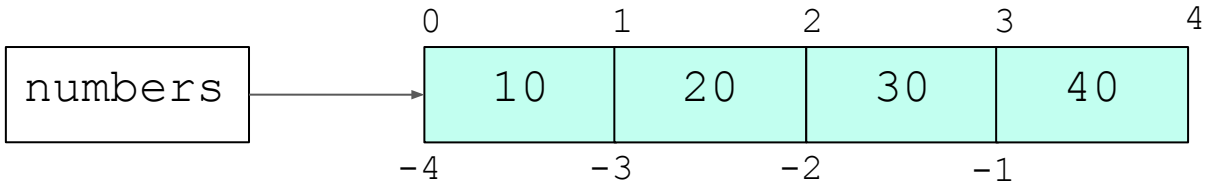
```
numbers[4]     #-> IndexError
numbers[-5]    #-> IndexError
```

Slicing

- We can extract a *subsequence* using **slicing**.

```
numbers[1:3]    #-> [20, 30]
numbers[0:4:2]  #-> [10, 30]  (step = 2)
numbers[2:2]    #-> []
```

Play ▸



- Negative indices may be used too.

```
numbers[-4:-2]  #-> [10, 20]
numbers[1:-1]   #-> [20, 30]
```

- Indices may be omitted for the start or end.

```
numbers[:2]  #-> [10, 20]
numbers[3:]  #-> [40]
numbers[:]   # a full copy of numbers
```

Traversing

- The most common way to traverse the elements of a sequence is with a **`for`** loop.

```
for f in fruits:
    print(f)
```

banana
pear
orange

Play ▸

- We may also traverse a sequence using the indexes.

```
for i in range(len(fruits)):
    print(i, fruits[i])
```

- In this case, we may use a **`while`** loop instead.

```
i = 0
while i < len(numbers):
    print(i, fruits[i])
    i += 1
```

- Or traverse the indexes and items simultaneously.

```
for i, f in enumerate(fruits):
    print(i, f)
```

Sequence operations

- The + operator concatenates and * repeats sequences.

```
s = [1, 2, 3] + [7, 7]  #-> [1, 2, 3, 7, 7]
s2 = [1, 2, 3] * 2      #-> [1, 2, 3, 1, 2, 3]
s3 = 3*[0]              #-> [0, 0, 0]
```

Play ▸

- Operator **`in`** checks if an element is included in the sequence. Operator **`not in`** means the opposite.

```
7 in s        #-> True
4 not in s    #-> True
```

- Some built-in functions apply to sequences.

```
sum(s)    #-> 20
min(s)    #-> 1
max(s)    #-> 7
```

Common sequence operations (Python documentation)

Lists are mutable

- Lists are **mutable**, i.e., we can change their contents.

```
numbers[1] = 99
numbers      #-> [10, 99, 20, 40]
```

Play ▸

- We can even change a sublist.

```
numbers[2:3] = [98, 97]
numbers      #-> [10, 99, 98, 97, 40]
```

- Lists have several methods to change their contents.

```
lst = [1, 2]
lst.append(3)        # appends 3 to end of lst → [1, 2, 3]
x = lst.pop()        # lst → [1, 2], x → 3
lst.extend([4, 5])   # lst → [1, 2, 4, 5]
lst.insert(1, 6)     # lst → [1, 6, 2, 4, 5]
x = lst.pop(0)       # lst → [6, 2, 4, 5], x → 1
```

List methods

- List objects have several useful methods.

Play ▸

`lst.append(item)`	Add item to the end
`lst.insert(pos, item)`	Insert item at the given position
`lst.extend(collection)`	Add all the items in the argument
`lst.pop()`	Remove last item
`lst.pop(pos)`	Remove item in given position
`lst.remove(item)`	Remove first occurence of given item (if any)
`lst.sort()`	Sort the items in the list
`lst.reverse()`	Reverse the order of items in the list
`lst.index(item)`	Position of first occurrence of given item
`lst.count(item)`	Number of occurrences of given item

Mutable sequence operations (Python documentation)

Exercises 1

- Do these codecheck exercises.

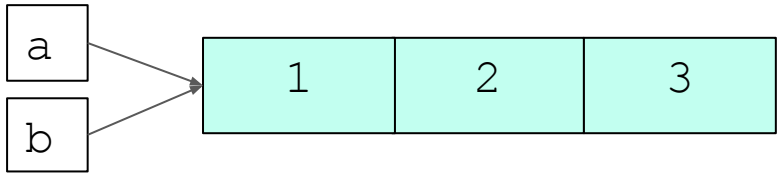
https://codecheck.io/private/resume/2210191825cgf8df7sukdtuafou4r6nbv7f/ybyx-qyhe-uzap-ames/58VQAF8TTYZCDQT19CEWZ57X4

Mutability and aliasing

- In Python, variables store ***references*** to objects.

Play ▸

```
a = [1, 2, 3]
b = a
# a and b refer to the same object!
# In other words, a and b are aliases.
```



- We can confirm that `a` and `b` refer to *the same* object.

```
a is b          #-> True
```

- If object is changed under one name, it is changed under all names! *This may seem strange, at first*.

```
b[0] = 9  # object referenced by b is modified
b         #-> [9, 2, 3] of course!
a         #-> [9, 2, 3] even though we did not change a!
```

- Aliasing = Different names referring to the same object.

Aliasing and argument passing

- Aliasing occurs whenever we pass objects as arguments.
- If the function changes the object, this reflects outside, too.

Play ▸

```
def grow(lst):
    lst.append(3)
    return lst[0] + lst[-1]
lst1 = [1, 2]
x = grow(lst1)
print(x, lst1)  # What's the output?
```

- This is memory-efficient and can be very useful.
- But if you don’t want it, just make a copy before changing.

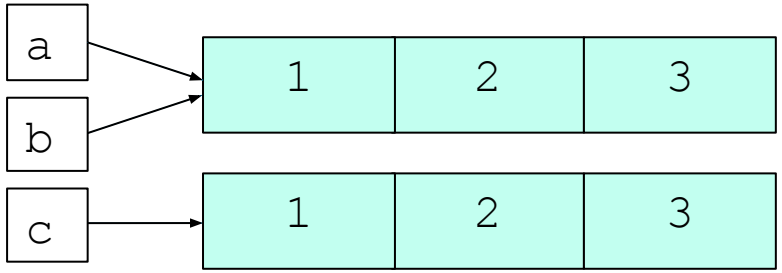
```
def grow(lst):
    r = lst[:]
    r.append(3)
    return r[0] + r[-1]
```

Equality *versus* identity

- Objects may be *equal* without being *the same*!

Play ▸

```
a = [1, 2, 3]
b = a
c = a[:]
```



- We test **equality** with **==** (or **!=**).

```
a == b   #-> True          a != b   #-> False
a == c   #-> True          a != c   #-> False
```

- We test **identity** with **is** (or **is not**).

```
a is b   #-> True          a is not b   #-> False
a is c   #-> False         a is not c   #-> True
```

- Identity *implies* equality!
- Equality *does not imply* identity.

Identity and immutable types

- Don't use **`is`** when you mean **`==`** !

```
[1, 2] == [1, 2]          #-> True
[1, 2] is [1, 2]          #-> False
"abx"[:2] == "ab"         #-> True
"abx"[:2] is "ab"         #-> False (probably...)
1000+1 is 1001            #-> False (probably...)
```

- For some immutable types, Python can <u>sometimes</u> detect equal values and share the same object to save space.

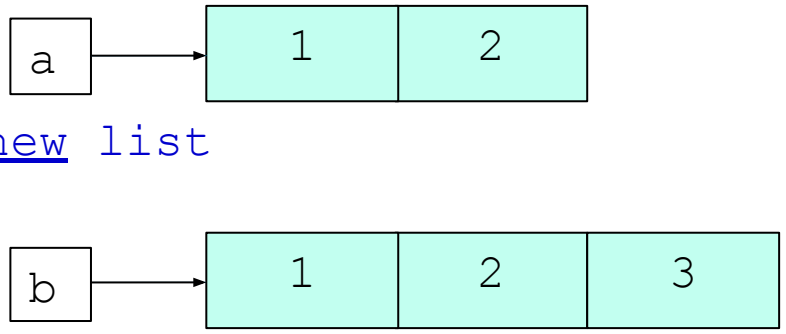
```
"ab" is "ab"  #-> probably True, but ...
10+1 is 11    #-> probably True, but ...
```

- This is implementation-dependent, so <u>do not rely on it</u>!

Cloning

- Sometimes, we need to make a copy of an object, so we can change it without changing the original.
- To clone lists, we may use the slicing operator `[:]`.

```
a = [1, 2]
b = a[:]    # slicing creates a new list
b is a      #-> False
b.append(3)
```



- We could also use the more general **`copy`** method.

```
b = a.copy()  # clone a
b is a        #-> False
```

- Other mutable types (such as sets and dictionaries) also have a `copy` method.
- Immutable types (tuples, strings) don't need one.

Exercises 2

- Do these codecheck exercises.

https://codecheck.io/private/resume/22101919141tgpzpw0buwe2rzpzc6dcxz0w/daky-ajiz-dohe-orog/AK1ZKMUEQ6Z2YD1L9H8U6RIZ0

Strings

- Strings are sequences of characters.
- String literals are delimited by single or double quotes.

```
fruit = 'orange'
```

- Like other sequences, we can use indexing and slicing.

```
letter = fruit[0] #-> 'o' (1st character)
len(fruit)        #-> 6   (length of string)
fruit[1:4]        #-> 'ran'
fruit[:-1]        #-> 'orang'
fruit[::-1]       #-> 'egnaro'
```

- We can also concatenate and repeat strings.

```
name = 'tom' + 'cat'   #-> 'tomcat'
gps = 2 * 'tom'        #-> 'tomtom'
```

Strings are immutable

- Unlike lists, strings in *Python* are **immutable**. Once a string is created it can't be modified.

```
fruit[0] = 'a'          #-> TypeError
```

- But we can create new strings by combining existing ones.

```
ape = fruit[:-1] + 'utan'    #-> 'orangutan'
```

- Even methods that imply modification actually only return a new string object.

```
fruit.upper()              #-> 'ORANGE'
fruit.replace('a', 'A')    #-> 'orAnge'
fruit                      #-> 'orange' (not changed)
```

String - traversal

- One way to traverse strings is with a `for` loop:

```
fruit = 'banana'
for char in fruit:
    print(char)
```

- Another way:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

- Another example:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
    print(letter + suffix)
```

Examples

- The following program counts the number of times the letter `'a'` appears in a string:

```
word                          =                          'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

- This prints the common characters in two strings. (For strings, the **`in`** operator returns `True` iff the first string appears as a substring in the second.)

```
for letter in word1:
    if letter in word2:
        print(letter)
```

More on strings

- The relational operators work on strings and other sequences.

```
if word < 'banana':
    print(word, 'comes before banana.')
elif word > 'banana':
    print(word, 'comes after banana.')
else:
    print('the same')
```

- Characters (letters, digits, punctuation) are stored as numeric codes (according to Unicode).
  - `ord(c)` - returns the code of the character `c`.
  - `chr(n)` - returns the character represented by code `n`.
- The `str` class has various built-in methods for checking for different classes of characters (`isalpha`, …).

String methods

- Strings have a lot of useful methods.

Play ▸

`str.isalpha()`
`str.isdigit()`
`str.is...`
	True if all characters are alphabetic.
	True if all characters are digits.
	...

`str.upper()`
`str.lower()`
`...`
	Convert to uppercase.
	Convert to lowercase.
	...

`str.strip()`
`str.lstrip()`
`str.rstrip()`
	Remove leading and trailing whitespace.
	Remove leading whitespace.
	Remove trailing whitespace.

`str.split()`	Split str by the whitespace characters.
`str.split(sep)`	Split str using sep as the delimiter.
`sep.join(lst)`	Join the strings in lst using delimiter sep.

String methods (Python documentation).

Tuples

- A **tuple** is an immutable sequence of values of any type.
- The values are indexed by integers, like in lists. The important difference is that **tuples are immutable**.
- Syntactically, a tuple is a comma-separated list of values.

```
t = 'a', 'b', 'c', 'd', 'e'
```

- It is common (and sometimes necessary) to enclose tuples in parentheses.

```
t = ('a', 'b', 'c', 'd', 'e')
```

- To create a tuple with a single element, you have to include a final comma:

```
t1 = ('a',)
type(t1)    #->  <type 'tuple'>
```

Tuples (2)

- Another way to create a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple:

```
t = tuple()          # t → ()
```

- If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

```
t = tuple('ape')     # t → ('a', 'p', 'e')
t = tuple([1, 2])    # t → (1, 2)
```

- Most list operators also work on tuples.
- We can't modify the elements in a tuple, but we can replace one tuple with another.

```
t = t + (3, 4)           # t → (1, 2, 3, 4)
```

Exercises 3

- Do these codecheck exercises.

https://codecheck.io/private/resume/22101920304vo52c01hatsh8iexmlm3xkib/ynes-ywuz-apuz-uzyk/C119NV996IFJLH65J24M8P5EI

Zipping and Enumerating

- The built-in function `zip` takes two or more sequences and generates a sequence of tuples, each containing one element from each sequence.

```
s = 'abc'
t = [4, 3, 2]
list(zip(s, t))  # → [('a', 4), ('b', 3), ('c', 2)]
```

- `enumerate` generates a sequence of (index, item) pairs.

```
enumerate('abc') # → (0, 'a'), (1, 'b'), (2, 'c')
```

- You can use tuple assignment in a **`for`** loop to traverse a sequence of tuples:

```
s = 'somestuff'
for i, c in enumerate(s):
    print(i, c)
```

Play ▸

Sequence ordering

- The **relational operators** work with sequences:
  - Python starts by comparing the first element from each sequence.
  - If they are equal, it compares the next elements and repeats.
  - When the elements differ, it returns the result of their comparison.

```
"aba" < "abel"     #-> True

(0, 1, 2) < (0, 3, 4)     #-> True
```

- The `sorted` function and the list `sort` method work the same way. They sort primarily by first element, but in the case of a tie, they sort by second element, and so on.