Fundamentos de Programação

António J. R. Neves

João Rodrigues

Departamento de Electrónica, Telecomunicações e Informática

Universidade de Aveiro

`an@ua.pt / jmr@ua.pt`

`http://elearning.ua.pt/`

Summary

- Sequence types
  - Lists
  - Strings

Lists

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

```
numbers = [10, 20, 30, 40]
fruits = ['banana', 'pear', 'orange']
things = ['spam', 2.0, 5, [1, 2]]  # a list inside!
empty = []       # an empty list
```

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

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

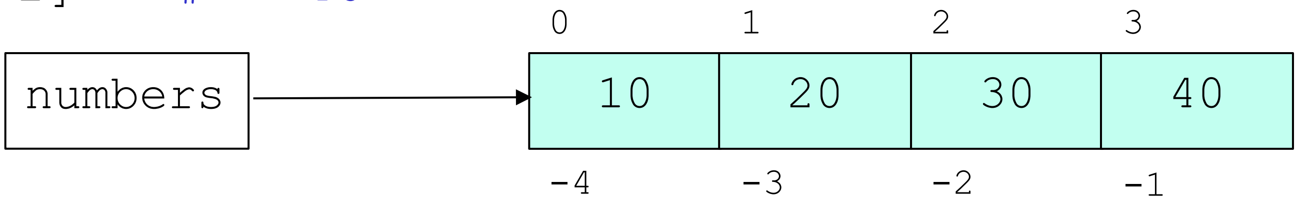
Indexing

- We can be 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'
```

- A negative index counts backward from the end.

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



- Any integer expression may be used as an index.

```
numbers[(6+1)%4]    #-> 40
```

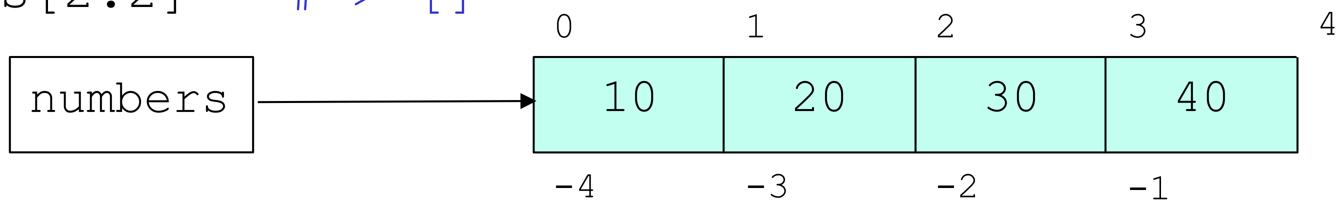
- Using an index outside the list bounds is an error.

```
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]    #-> []
```



- 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
```

Lists are mutable

- Lists are **mutable**, i.e., we can change its elements.

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

- We can even change a sublist.

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

- The contents change, but the object is the same.

```
a = b = [1, 2, 3] # a and b refer to the same object
a[0] = 9          # object a is modified
b                 #-> [9, 2, 3]
```

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

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

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
```

- But sometimes we use the indices, e.g., when updating.

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

- In this case, we could have used a `while` loop instead.

Lists – Operations and methods

- The + operator concatenates and * repeats sequences.

```
s = [1, 2, 3] + [7, 8]   #-> [1, 2, 3, 7, 8]
z = [0]*3                #-> [0, 0, 0]
```

- Lists have several useful methods.

```
z.append(3)   # appends 3 to end of z -> [0, 0, 0, 3]
z.extend([2, 1]) # appends elements of list to z
x = s.pop()   # s -> [1, 2, 3, 7], x -> 8
```

- 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
```

Lists – more operations

- If we know the index of the element to delete, we can use `pop` - it modifies the list and returns the element that was removed.
- If we don't need the removed value, we can use the `del` operator.
- If we know the element to remove (but not the index), we can use `remove`.
- To remove more than one element, we can use `del` with a slice index.
- `sort` arranges the elements of the list from low to high.
- `sum` adds all the elements of a list.

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)
```

- 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.

```
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 in python3).
- `ord(c)` - returns the code of the character.
- `chr(n)` - returns character represented code `n`.
- String class has various built-in methods which allows to check for different types of strings (`isalpha`, …).