Programação I / Introdução à Programação

Capítulo 5, "Data types" (cadeias de caracteres, "strings")

João Pedro Pedroso

2024/2025

Última aula:

- Tipos de dados: *Strings*

Hoje:

- *Strings:*
  - Detalhes
  - Exemplos de utilização

Comparação lexicográfica

Feita elemento a elemento, para valores do tipo sequência

- A comparação entre *strings* é feita da seguinte forma:
  - duas *strings* são iguais se:
    - têm o mesmo comprimento
    - cada par de elementos correspondentes são iguais
  - duas *strings* são ordenadas de acordo com o primeiro elemento diferente
    - por exemplo, o valor de `"abcXa" < "abcYz"` é `True` porque `"X" < "Y"` também é `True`
  - se o elemento correspondente não existe numa das *strings*, a *string* mais curta é ordenada primeiro
    - `"abc" < "abc*"` é `True`, qualquer que seja o carater `"*"`
- A comparação entre outros tipos de dados envolvendo sequências (e.g., listas e tuplos) é feita da mesma forma
- Sequências de tipos diferentes (e.g., listas e tuplos) são sempre diferentes

Testar ocorrência

`txt1 in txt2` → testa ocorrência de `txt1` dentro de `txt2`

```
>>> 'ana' in 'Banana'
True
>>> 'mana' in 'Banana'
False
>>> 'mana' not in 'Banana'
True
>>> 'Banana' in 'Banana'
True
>>> '' in 'Banana'
True
```

Padrão para criar *strings* com base noutras

```
def remove_vowels(phrase):
    vowels = "aeiou"
    string_sans_vowels = ""
    for letter in phrase:
        if letter.lower() not in vowels:
            string_sans_vowels += letter
    return string_sans_vowels
```

Padrão para criar *strings* com base noutras

```
def remove_vowels(phrase):
    vowels = "aeiou"
    string_sans_vowels = ""
    for letter in phrase:
        if letter.lower() not in vowels:
            string_sans_vowels += letter
    return string_sans_vowels
```

```
>>> remove_vowels("Programacao I")
'Prgrmc '
```

Procurar dentro de uma *string*

```
def my_find(haystack, needle):
    for index, letter in enumerate(haystack):
        if letter == needle:
            return index
    return -1
```

- `find` → inverso da indexação
- quando não se encontra o que se procura: retorna -1
- este idioma: "short-circuit evaluation"
  - logo que se encontra o que se procura, para-se a pesquisa

```
>>> haystack = "Bananarama!"
>>> print(my_find(haystack,'a'))
1
>>> print(haystack.find('a'))
1
```

Parâmetros opcionais

- Podemos modificar a função `my_find` para utilizar o índice onde deve começar a procurar:

```
1  def my_find(haystack, needle, start=0):
2      for index, letter in enumerate(haystack[start:]):
3          if letter == needle:
4              return index + start
5      return -1
```

- Podemos dar um valor "por omissão"
  - na definição de uma função `start=0`
  - se chamarmos `my_find` com dois argumentos, o valor de `start` será zero
  - na implementação do método `find` de *strings*, há dois parâmetros opcionais:

    `start` onde se começa a procurar
    `end` onde se termina a procura
    `"banana".find('a', 2, 5)` → 3

Mais métodos úteis para *strings*

find índice da primeira ocorrência

- `"abcde".find("cd")` → 2
- `"abcde".find("C")` → -1

replace substituir ocorrências

- `"banana".replace("ana", "ANA")` → `'bANAna'`
- `"banana".replace("ananas", "ANA")` → `'banana'`

upper substituir letras minúsculas por maiúsculas

- `"banana".upper()` → `'BANANA'`

lower substituir letras maiúsculas por minúsculas

- `"BANANA".lower()` → `'banana'`

split divite uma *string* com várias palavras numa lista de palavras

- `"banana".split("a")` → `['b', 'n', 'n', '']`
- `"where is the banana".split()` → `['where', 'is', 'the', 'banana']`

```
>>> phrase = """Well I never
... did
... said Alice"""
>>> print(phrase)
Well I never
did
said Alice
>>> words = phrase.split()    # by default, words separated by any white space
>>> words
['Well', 'I', 'never', 'did', 'said', 'Alice']
```

"Limpeza" de *strings*

- Muitas vezes, é útil remover todos os carateres que não são letras:

```
1  punctuation = "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"
2  def remove_punctuation(phrase):
3      phrase_sans_punct = ""
4      for letter in phrase:
5          if letter not in punctuation:
6              phrase_sans_punct += letter
7      return phrase_sans_punct
```

- Em vez de definirmos `punctuation`, podemos usar uma definição do módulo `string`:

```
1  import string
2  def remove_punctuation(phrase):
3      phrase_sans_punct = ""
4      for letter in phrase:
5      if letter not in string.punctuation:
6          phrase_sans_punct += letter
7      return phrase_sans_punct
```

Ver mais atributos na documentação do Python:
`https://docs.python.org/3/library/string.html`

Módulo string

`string.`**`ascii_letters`**

The concatenation of the `ascii_lowercase` and `ascii_uppercase` constants described below. This value is not locale-dependent.

`string.`**`ascii_lowercase`**

The lowercase letters `'abcdefghijklmnopqrstuvwxyz'`. This value is not locale-dependent and will not change.

`string.`**`ascii_uppercase`**

The uppercase letters `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. This value is not locale-dependent and will not change.

`string.`**`digits`**

The string `'0123456789'`.

`string.`**`hexdigits`**

The string `'0123456789abcdefABCDEF'`.

`string.`**`octdigits`**

The string `'01234567'`.

`string.`**`punctuation`**

String of ASCII characters which are considered punctuation characters in the `c` locale: `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`.

`string.`**`printable`**

String of ASCII characters which are considered printable. This is a combination of `digits`, `ascii_letters`, `punctuation`, and `whitespace`.

Códigos ASCII

- Códigos numéricos associados aos carateres
- Estabelecem a ordem lexicográfica

32	33 !	34 "	35 #	36 $	37 %	38 &	39 '
40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127

Notas

- ASCII foi desenvolvido apenas tendo em consideração a língua inglesa
- Durante muitos anos cada país/língua tinha uma extensão, tipicamente de 128 a 255
- Não se podia escrever um texto em mais do que uma língua...
- Unicode
  - standard para ultrapassar esta limitação
    Everyone in the world should be able to use their own language on phones and computers
  - objetivo: ter uma representação para qualquer carater usado em línguas humanas
  - está a ser continuamente revisto e atualizado
  - inclui os carateres ASCII com a mesma codificação
  - ver `https://www.unicode.org`
  - versões atuais do Python (>3) usam carateres Unicode

Códigos unicode

- Extensão "moderna" da tabela ASCII
- Incluem carateres acentuados, emoji, ...

```
0061    'a'; LATIN SMALL LETTER A
0062    'b'; LATIN SMALL LETTER B
0063    'c'; LATIN SMALL LETTER C
...
007B    '{'; LEFT CURLY BRACKET
...
2167    'Ⅷ'; ROMAN NUMERAL EIGHT
2168    'Ⅸ'; ROMAN NUMERAL NINE
...
265E    '♞'; BLACK CHESS KNIGHT
265F    '♟'; BLACK CHESS PAWN
...
1F600   '😀'; GRINNING FACE
1F609   '😉'; WINKING FACE
...
```

Funções ord e chr

- `ord(c)` → devolve o código númerico associado a `c`
- `chr(n)` → devolve o carater associado ao código `n`

```
>>> ord('A')
65
>>> ord('a')
97
>>> ord('b')
98
>>> chr(98)
'b'
>>> chr(0x1F607)
'??'     # <--  check which character is this
>>>
```

Exemplo: a cifra de César

- Utilizada pelo imperador Júlio César (100~AC–44~AC)
- Um dos métodos mais simples para codificar um texto
- Cada letra é substituida pela que dista k posições no alfabeto
- Quando ultrapassa a letra 'z', volta à letra 'a'

Exemplo: para $k = 3$, a rotação é:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
d e f g h i j k l m n o p q r s t u v w x y z a b c
```

Logo: "`ataque`" é codificado como "`dwdtxh`"

Recorde: códigos de caracteres

Funções pré-definidas para converter entre caracteres e códigos numéricos:

- `ord(c)` obter o código numérico de um carater `c`
- `chr(n)` obter o carater com código `n`

Exemplos:

```
>>> ord('A')
65
>>> chr(66)
'B'
```

Rodar um carater

- Definimos uma função para deslocar as letras minúsculas k posições
- Outros caracteres ficam inalterados

```
def rodar(k,c):
    "Rodar o carater c por k posições."
    if c>='a' and c<='z':
       n = ord(c)-ord('a')
       return chr((n+k)%26 + ord('a'))
    else:
       return c
```

Cifrar um texto

- Para cifrar, percorremos o texto e aplicamos "rodar" a cada caracter.

```
1  def cifrar(k, txt):
2      "Cifrar txt rodando k posições."
3      msg = ''    # mensagem cifrada
4      for c in txt:
5          msg += rodar(k,c)
6      return msg
```

- Exemplo:

```
>>> cifrar(3, "ataque de madrugada!")
'dwdtxh gh pdguxjdgd!'
```

Para descodificar, basta cifrar com deslocamento simétrico:

```
>>> cifrar(-3, "dwdtxh gh pdguxjdgd!")
'ataque de madrugada!'
```

Caracteres especiais

- Numa cadeia de caracteres a barra \ serve para dar indicação de caracteres de *escape*
- Estes caracteres são interpretados
  - e.g., pelo comando `print`
  - servem para representar caracteres especiais
    - tabulação `\t`
    - mudança de linha `\n`
    - ...
- Exemplos:
  - tabulação

```
>>> print('um\tdois')
um      dois
```

  - mudança de linha

```
>>> print('um\ndois')
um
dois
```

- Noções estudadas
  - strings
  - métodos: `lower`, `upper`, `find`, `replace` ...
  - `ord()`, `chr()`
  - ASCII, unicode
- Próxima aula
  - strings: formatação