Programação I / Introdução à Programação

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

João Pedro Pedroso

2023/2024

Última aula:

- Funções

Hoje:

- Typos de dados: *Strings*

Algumas notas

- Escrever um programa não resolve problemas!
  - apenas *automatiza* os passos necessários
  - a resolução tem de ser pensada antes disso...
    $\rightarrow$ usar papel e lápis antes de começar a programar
- Não utilizar `print` e `input` dentro de funções
  - exceções:
    - quando o papel dessa função é input/output
    - quando se está a fazer *debugging* $\rightarrow$ remover *print*'s depois de o código estar corrigido

Revisões: números pseudo-aleatórios

- Necessários em muitas situações:
  - num jogo, para lançar um dado/moeda ao ar;
  - baralhar um conjunto de cartas;
  - escolher um exercício para examinar um aluno;
  - simular o tempo de espera por um autocarro;
  - codificar informação numa consulta à conta bancária;
  - ...
- módulo `random` tem funcionalidades para isto, e.g.:

  `random.random()` $\rightarrow$ `float` no intervalo $[0, 1[$

  `random.choice(xs)` $\rightarrow$ um valor (aleatório) da lista `xs`
  - `xs` não pode ser vazia

  `random.randrange(a,b[,step])` $\rightarrow$ inteiro aleatório entre `a` e `b`, com distribuição uniforme
  - limite superior não é incluído (tal como `range`)

  `random.randint(a, b)` $\rightarrow$ inteiro N aleatório, $a \leq N \leq b$
  - alias para `randrange(a, b+1)`

- os geradores de números pseudo-aleatórios são baseados num algoritmo determinístico
- são inicializados com uma semente (*seed*)
  - inicialização "manual": e.g., `seed(1)` → comportamento sempre igual
  - inicialização "por omissão": em geral será diferente de cada vez que se executa o programa
    - por isso, o comportamento, na prática, é aleatório

```
import random
# random.seed(1) # uncomment this line for having always the same numbers gener
prob = random.random()
print(prob)
diceThrow = random.randint(1,6)
print(diceThrow)
```

output:

```
0.4079410636000885
6
```

- *Nota:* nos *slides* anteriores utilizamos um gerador disponível *"por omissão"*
- Podemos criar vários geradores independentes:

```
>>> import random
>>> rng_a = random.Random(1)
>>> rng_b = random.Random(1)
>>> rng_a.random()
0.13436424411240122
>>> rng_a.random()
0.8474337369372327
>>> rng_a.random()
0.763774618976614
>>> rng_b.random()
0.13436424411240122
>>> rng_b.random()
0.8474337369372327
>>> rng_b.random()
0.763774618976614
>>> rng_a.random()
0.2550690257394217
```

Aula de hoje:

- Tipos de dados compostos
- Strings
- Codificação ASCII (próxima aula: Unicode)

Tipos de dados compostos

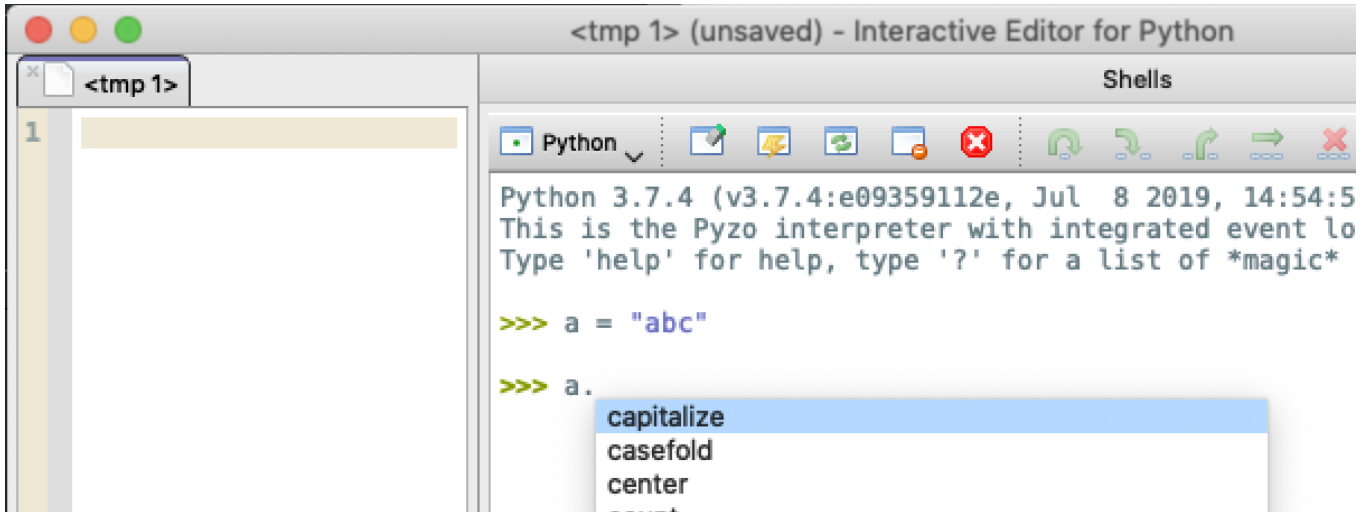
- Vimos já alguns tipos de dados do Python
  - `int, float, bool, str`
  - vimos também listas e tuplos
  - `str`: tipo de dados usado para representar palavras escritas
- *Strings*, listas e tuplos são diferentes dos tipos anteriores:
  - são constituidos por elementos mais pequenos
  - *strings* (cadeias de carateres):
    - cada um desses elementos mais pequenos é uma *string* com um carater
  - são chamados tipos de dados compostos
  - por vezes interessa-nos vê-los como um todo, outras vezes queremos aceder a cada uma das partes

Strings como objetos simples

- Tal como vimos com as tartarugas, *strings* são objetos com *atributos* e *métodos*

```
>>> a = "Hello"
>>> b = ", "
>>> c = "World!"
>>> our_string = a + b + c    # our_string = "Hello, World!"
>>> all_caps = our_string.upper()
>>> all_caps
'HELLO, WORLD!'
>>> our_string
'Hello, World!'
```

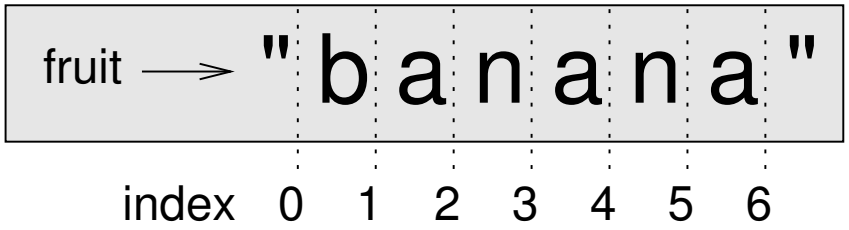
- Outros métodos: `lower`, `capitalize`, `swapcase`, ...
  Podemos consultar o manual, listá-los com `help(str)`, ou:



Strings como objetos compostos

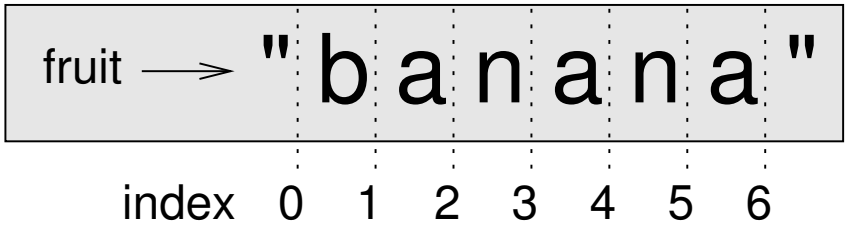
- Podemos aceder a cada uma das partes de uma *string* com o operador de indexação, []

```
>>> fruit = "banana"
>>> len(fruit)   # length (number of elements)
6
>>> letter = fruit[1]    # 1 -> index
>>> print(letter)
'a'
```



- Indexação de *strings*, de listas e de tuplos é semelhante

Strings: indexação



- Primeiro elemento ("b") → índice zero
- Índice → qualquer expressão com valor inteiro

```
>>> fruit = "banana"
>>> fruit[0]
'b'
>>> fruit[1]
'a'
>>> fruit[2*2+1]
'a'
>>> fruit[-1]
'a'
>>> fruit[-2]
'n'
>>> fruit[-3]
'a'
```

Strings: indexação com valores negativos

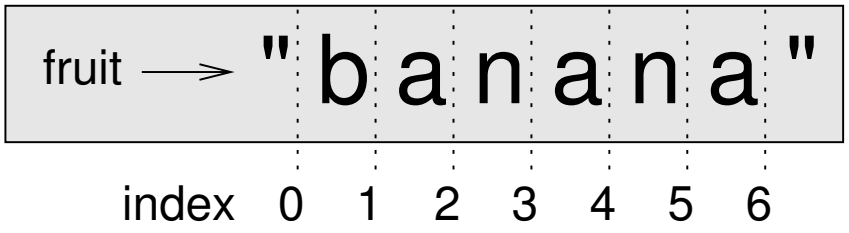
```
>>> word = "banana"
>>> size = len(word)
>>> word[size]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Último elemento da *string* `word`:

- índice `[size - 1]`
  - `word[size-1]` → `"a"`
- índice `[-1]` → *primeiro elemento a contar do fim:*
  - `word[-1]` → `"a"`
- as duas formas são equivalentes; em Python, normalmente usa-se a segunda

```
>>> word[-2]
'n'
```

Palavra reservada `enumerate`



- Podemos visualizar os índices com `enumerate`:

```
>>> fruit = "banana"
>>> list(enumerate(fruit))
[(0, 'b'), (1, 'a'), (2, 'n'), (3, 'a'), (4, 'n'), (5, 'a')
```

Fatias

Permitem obter *substrings*

```
>>> phrase = "Pirates of the Caribbean"
>>> print(phrase[0:5])
Pirat
>>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
>>> friends[2:4]
['Brad', 'Angelina']
```

`txt[i:j]`	*substring* de `txt` entre indices `i` e `j-1` inclusive
`txt[i:]`	*substring* desde o índice `i` até ao final
`txt[:j]`	*substring* desde o ínicio até ao índice `j-1` inclusive

Nota: o valor de `j` pode ser superior ao tamanho da string, no acesso a uma fatia isso não dá erro (resulta na *string* desde *i* até ao final)

Percorrer *strings*

- Semelhante ao idioma que vimos para percorrer listas:

```
1  word="Banana"
2  for letter in word:
3      print(letter)
```

```
B
a
n
a
n
a
```

Outros idiomas

- Básico:

```
1  word="Banana"
2  for letter in word:
3      print(letter)
```

- Com indexação

```
1  for i in range(len(word)):
2      print(i, word[i])
```

- Com indexação e ciclo while (pouco recomendado)

```
1  i = 0
2  while i < len(word):
3      letter = word[i]
4      print(i, letter)
5      i += 1
```

- Com tuplos fornecidos por `enumerate`
  - usado quando precisamos do índice e do valor e cada elemento

```
1  for i,letter in enumerate(word):
2      print(i, letter)
```

Comparação de *strings*

- Operadores de comparação funcionam com *strings*:

```
>>> word = "banana"
>>> word == "apple"
False
>>> word == "banana"
True
>>> word == "Banana"
False
```

- É usada uma ordem *lexicográfica;* ordenação não é alfabética!

```
1  word = "Zebra"
2  if word < "banana":
3      print("Your word, " + word + ", comes before banana.")
4  elif word > "banana":
5      print("Your word, " + word + ", comes after banana.")
6  else:
7      print("Well, we have no bananas...")
```

```
Your word, Zebra, comes before banana.
```

Códigos ASCII

In the Beginning, there was ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SPC	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

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 ~	

Em Python

- Podemos saber o índice na tabela ASCII com `ord`:
- Podemos saber carater de um índice na tabela ASCII com `chr`:

```
>>> ord("a")
97
>>> ord("b")
98
>>> ord("A")
65
>>> ord("Z")
90
>>> chr(90)
'Z'
>>> chr(97)
'a'
```

Em Python: comparação lexicográfica

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 `"abcX" <= "abcY"` 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 `"*"`

Em Python

```
"Joao" < "Jose"
```

↓

```
"J" < "J"        74 == 74
"o" < "o"        111 == 111
"a" < "s"        97 < 115      comparação resulta em True
"o" < "e"
```

Em Python

```
"Joao" < "Jose"
```

↓

```
"J" < "J"        74 == 74
"o" < "o"        111 == 111
"a" < "s"        97 < 115      comparação resulta em True
"o" < "e"
```

```
>>> "Joao" < "Jose"
True
```

Em Python

```
"Jose" < "Joao"
```

↓

```
"J" < "J"        74 == 74
"o" < "o"        111 == 111
"s" < "a"        115 > 97        X  comparação falha, resulta e
"e" < "o"
```

Em Python

```
"Jose" < "Joao"
```

↓

```
"J" < "J"        74 == 74
"o" < "o"        111 == 111
"s" < "a"        115 > 97        X  comparação falha, resulta e
"e" < "o"
```

Explique:

```
>>> "Joao" < "Joao"
False
```

Strings e mutabilidade

- As *strings* são imutáveis: não as podemos alterar

```
>>> greeting = "Jello, world!"
>>> greeting[0] = 'H'
TypeError: 'str' object does not support item assignment
```

- mas podemos criar novas *strings*:

```
>>> new_greeting = "H" + greeting[1:]
>>> new_greeting
'Hello, world!'
```

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
```

Noções estudadas

indexing	`[]`
length function	`len`
for loop traversal	`for`
slicing	`[:]`
string comparison	`>, <, >=, <=, ==, !=`
membership operators	`in`, `and`, `not`, `in`
	`chr`, `ord`

Próxima aula

- Unicode

```
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
...
```

- Tipos de dados do Python: *strings* (conclusão)