

Programação e Base de Dados /Computação para Detecção Remota

Introdução à linguagem Python

Sérgio Crisóstomo,
(adaptado dos slides de Rita Ribeiro,
Pedro Vasconcelos e João Pedro Pedroso)
2022/2023

Departamento de Ciência de Computadores



1. Ciclos for
2. Ciclos while
3. Saída e continuação num ciclo
4. Cadeias de caracteres
5. Exemplo: a cifra de César
6. Caracteres especiais

- Repetir instruções um número variável de vezes
- Para exprimir computação, tabelar uma função, etc.

Ciclos for

```
for variável in sequência:  
    ⋮  
    instruções a repetir  
    ⋮
```

- Repete instruções com a variável a tomar sucessivos valores da sequência
- O número de iterações está limitado pelo comprimento da sequência

Exemplo

```
def crescente(n):  
    # i vai de 1 até n-1 por incrementos de 1  
    for i in range(1, n):  
        print(i)  
    # fim do ciclo  
    print('fim')
```

Exemplo (cont.)

```
>>> crescente(10)
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
fim
```

Sequências aritméticas

A função `range(a, b, k)` gera sequências em progressão aritmética

$$a, a + k, a + 2k, \dots$$

cujos valores são menores que b .

```
>>> list(range(10))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
>>> list(range(1, 11))  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
>>> list(range(0, 30, 5))  
[0, 5, 10, 15, 20, 25]
```


Ciclos while

Ciclos while

```
while condição:  
    :  
    instruções a repetir  
    :
```

- Repete as instruções enquanto a condição for verdadeira
- A condição é re-avaliada após cada iteração
- O corpo do ciclo deve **modificar alguma variável** da condição

Exemplo

```
def crescente(n):  
    i = 1  
    while i < n:  
        # repete estas instruções  
        print(i)  
        i = i+1  
    # após o fim do ciclo  
    print('fim')
```

Exemplo (cont.)

```
>>> crescente(10)
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
fim
```

Modificar variáveis

Muitas vezes necessitamos de incrementar ou decrementar variáveis.

```
⋮  
i = i+1  
⋮
```

Esta atribuição pode ser abreviada:

```
⋮  
i += 1  
⋮
```

Modificar variáveis (cont.)

Mais geralmente, temos as seguintes atribuições abreviadas:

$$v \ += \ k \quad \longrightarrow \quad v \ = \ v \ + \ k$$
$$v \ -= \ k \quad \longrightarrow \quad v \ = \ v \ - \ k$$
$$v \ *= \ k \quad \longrightarrow \quad v \ = \ v \ * \ k$$
$$v \ /= \ k \quad \longrightarrow \quad v \ = \ v \ / \ k$$
$$v \ \ /= \ k \quad \longrightarrow \quad v \ = \ v \ /\ / \ k$$
$$v \ \ ** \ = \ k \quad \longrightarrow \quad v \ = \ v \ ** \ k$$

Um ciclo while pode não terminar

```
def repete():  
    n = 1  
    while True: # repete indefinidamente  
        print('Se',n,'elefantes incomodam muita gente')  
        print(n+1,'elefantes incomodam muito mais!')  
        n += 1
```

Sequência de Collatz

Nem sempre é fácil concluir que um ciclo while não termina.

Exemplo:

Calcular a Sequência de Collatz para um dado $n > 0$

Enquanto n não for igual a 1, gerar o próximo elemento da sequência usando a função:

$$f(n) = \begin{cases} n/2 & \text{se } n \text{ é par} \\ 3n + 1 & \text{se } n \text{ é ímpar} \end{cases}$$

Sequência de Collatz (cont.)

```
def sequencia(n):  
    print(n)  
    while n != 1:  
        if n%2 == 0:    # n é par  
            n = n//2  
        else:           # n é ímpar  
            n = 3*n+1  
    print(n)
```

Sequência de Collatz (cont.)

- `sequencia(1): 1`
- `sequencia(2): 2, 1`
- `sequencia(3): 3, 10, 5, 8, 4, 2, 1`
- `sequencia(4): 4, 2, 1`
- `sequencia(5): 5, 8, 4, 2, 1`
- `sequencia(6): 6, 3, 10, 5, 8, 4, 2, 1`
- ...
- `sequencia(27): 27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, ...` → são necessárias 111 iterações para chegar a 1
- ...

Não sabemos se esta função termina para todo $n > 0$.

http://en.wikipedia.org/wiki/Collatz_conjecture

Calcular o fatorial de n

$$0! = 1$$

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n$$

1. Inicialmente: $p = 1$
2. Repetir para i de 2 até n :
$$p \leftarrow p \times i$$
3. No fim do ciclo o valor de p é $n!$

Fatorial usando um ciclo for

```
def factorial(n):  
    # p acumula o produto 1*2*...*n  
    p = 1  
    # repetir para i de 2 até n  
    for i in range(2, n+1):  
        p *= i  
    # resultado é o valor de p  
    return p
```

Fatorial usando um ciclo while

```
def factorial(n):  
    # p acumula o produto 1*2*...*n  
    p = 1  
    # repetir para i de 2 até n  
    i = 2  
    while i<=n:  
        p *= i  
        i += 1  
    # resultado é o valor de p  
    return p
```

Fatorial usando um ciclo while (cont.)

Mais geralmente:

- Podemos *sempre* re-escrever um ciclo for num ciclo while.
- Nem sempre podemos re-escrever um ciclo while como um ciclo for (exemplo: a função que gera a sequência de Collatz).

Saída e continuação num ciclo

Duas instruções permitem alterar a execução de um ciclo:

break sair a meio do ciclo

continue passar à próxima iteração

Break com um ciclo for

```
for i in [12, 16, 17, 24, 29]:  
    if i % 2 == 1: # se é ímpar  
        break      # ... termina o ciclo  
    print(i)  
print("fim")
```

Resultado:

12
16
fim

Continue com um ciclo for

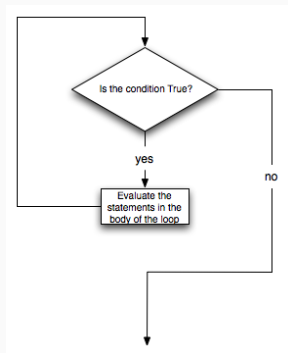
```
for i in [12, 16, 17, 24, 29, 30]:  
    if i % 2 == 1:      # se é impar  
        continue      # ...passa ao próximo  
    print(i)  
print("fim")
```

Resultado:

```
12  
16  
24  
30  
fim
```

Saída de um ciclo while

O teste do ciclo *while* ocorre **antes** da execução do corpo.



Podemos usar *break* para colocar um teste no corpo do ciclo.

Exemplo: teste no meio do corpo

```
total = 0
while True:
    resposta = input("Insira um número (ou vazio)")
    if resposta == ' ':
        break          # termina o ciclo
    total += int(resposta)
print("Total = ", total)
```

Que tipo de ciclo usar? (1)

while repetir **enquanto** a condição é verdadeira
for repetir **para** uma sequência de valores

Que tipo de ciclo usar? (2)

- Ciclo for**
1. iterar sobre sequências aritméticas;
 2. iterar sobre listas ou tuplos.

Ciclo while nos outros casos.

Cadeias de caracteres

- São sequências de caracteres
- Podemos tratá-las como uma entidade única
- Mas também podemos aceder aos caracteres individuais

Operações básicas

- comprimento

```
>>> len('Olá')  
3
```

- concatenação

```
>>> 'Olá'+'Mundo'  
'OláMundo'
```

- repetição

```
>>> 3*'Olá'  
'OláOláOlá'
```

- pertença

```
>>> 'l' in 'Olá'  
True
```

- iteração

```
>>> for c in 'Olá':  
        print(c)  
  
O  
l  
á
```

`txt` $\longrightarrow$ 'B | a | n | a | n | a'
 0 | 1 | 2 | 3 | 4 | 5

- Índices de 0 até `len(txt)-1`
- Caracteres: `txt[0]`, `txt[1]`, ...
- Último caracter: `txt[len(txt)-1]`
- Penúltimo caracter: `txt[len(txt)-2]`
- Índices negativos contam a partir do fim:

`txt[-1] == txt[len(txt)-1]`

`txt[-2] == txt[len(txt)-2]`

Exemplo

```
>>> txt = 'Banana'
>>> txt[0]
'B'
>>> txt[1]
'a'
>>> txt[2]
'n'
>>> len(txt)
6
```

Exemplo (cont.)

```
>>> txt = 'Banana'
>>> txt[5]
'a'
>>> txt[-1]
'a'
>>> txt[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
>>> txt[-6]
'B'
```

`txt[i:j]` sub-cadeia entre índices i e $j - 1$ inclusíve

`txt[i:]` sub-cadeia desde índice i até ao final

`txt[:j]` sub-cadeia desde o início até ao índice $j - 1$ inclusíve

```
>>> txt = 'Banana'
```

```
>>> txt[:3]
```

```
'Ban'
```

```
>>> txt[3:]
```

```
'ana'
```

```
>>> txt[2:5]
```

```
'nan'
```

Cadeias são imutáveis

- Não podemos modificar os caracteres duma cadeia
- Mas podemos construir uma outra cadeia

Exemplo

```
>>> txt = 'Bamana'
>>> txt[2] = 'n'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment

>>> txt = txt[:2]+'N'+txt[3:]
>>> txt
'BaNana'
```

Comparação de cadeias

`txt1 <= txt2` menor ou igual pela **ordem lexicográfica** (i.e. ordem do dicionário)

```
>>> 'abba' <= 'banana'
True
>>> 'abade' <= 'abacaxi'
False
>>> 'B' <= 'a'
True
>>> print(ord('B'), ord('a'))
66 97
```

A comparação é feita pelos *códigos numéricos* dos caracteres.

Códigos dos caracteres

- Cada caracter tem associado um código numérico num dado sistema de codificação (ASCII,ISO-LATIN-1,UTF8, etc.)
- Alguns códigos ASCII

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 ~	

Testar ocorrência

txt1 in txt2 testar ocorrência de txt1 dentro de txt2

```
>>> 'ana' in 'Banana'  
True
```

```
>>> 'mana' in 'Banana'  
False
```

```
>>> 'Banana' in 'Banana'  
True
```

```
>>> '' in 'Banana'  
True
```

Testar ocorrência (cont.)

txt1 not in txt2 testar a não ocorrência de `txt1` dentro de `txt2`

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

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

```
>>> 'Banana' not in 'Banana'  
False
```

```
>>> '' not in 'Banana'  
False
```

Percorrer uma cadeia (1)

Usando um ciclo sobre os índices.

```
for i in range(len(txt)):  
    # invariante:  $0 \leq i \leq \text{len}(\text{txt}) - 1$   
    c = txt[i]  
    print(c)
```

Percorrer uma cadeia (2)

Usando um ciclo sobre a sequência.

```
for c in txt:  
    print c
```

- Evita a necessidade da variável índice
- Mais geralmente, o ciclo `for` percorre quaisquer sequências

Procurar a primeira ocorrência

Vamos escrever uma função `primeira(c, txt)` que:

- procure se um caracter `c` ocorre na cadeia `txt`;
- em caso afirmativo, retorna o *índice* da primeira ocorrência;
- em caso negativo, retorne -1.

Exemplo

```
>>> primeira('n', 'banana')  
2  
>>> primeira('m', 'banana')  
-1
```

Procurar a primeira ocorrência

```
def primeira(c, txt):  
    "Procura a primeira ocorrência de c em txt."  
    for i in range(len(txt)):  
        if txt[i] == c: # encontrou  
            return i      # retorna o índice  
    # fim do ciclo  
    return -1             # não encontrou: retorna -1
```

Métodos sobre cadeias

- As cadeias são **objetos**
- Suportam várias **operações pré-definidas**
- Invocadas como **métodos**:

obj.método($arg_1, arg_2, \dots$)

Métodos sobre cadeias (cont.)

find índice da primeira ocorrência

replace substituir ocorrências

upper substituir letras minúsculas por maiúsculas

lower substituir letras maiúsculas por minúsculas

Exemplos:

```
>>> txt = "Banana"
>>> txt.find('ana')
1
>>> txt.replace('a','A')
'BAnAnA'
>>> txt.upper()
'BANANA'
```

Exemplo: a cifra de César

Exemplo: a cifra de César

- Um dos métodos mais simples para codificar um texto.
- Cada letra é substituída 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”.

Códigos de caracteres

Vamos usar duas funções pré-definidas para converter entre caracteres e códigos numéricos:

ord(c) obter o código numérico dum 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.

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

Cifrar um texto (cont.)

Exemplo:

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

Para decodificar, basta cifrar com deslocamento simétrico:

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


Caracteres especiais

Caracteres especiais

Numa cadeia de caracteres a barra `\` serve para dar indicação de caracteres de *escape*.

Estes caracteres são interpretados pelo comando `print` e servem para representar caracteres especiais.

Exemplos:

tabulação

```
>>> print('um \t dois')  
um      dois
```

mudança de linha

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