Fundamentos de Programação @ LEIC/LETI

Aulas 08 e 09

Listas

Listas. Método de passagem de parâmetros. Exemplos. O crivo de Eratóstenes. Algoritmos de procura e de ordenação.

Alberto Abad, Tagus Park, IST, 2018

Listas em Python

- Em Python, uma lista (`list`) é uma sequência de elementos (como os tuplos), mas como uma diferência fundamental: listas são **mutáveis**.
  - Alterar, eliminar, acrescentar elementos...

```
<lista> ::= [] | [<elementos>]
<elementos> ::= <elemento> | <elemento>, <elementos>
<elemento> ::= <expressão> | <tuplo> | <lista> | <dicionário>
```

- Listas de um elemento: `[e]` (não há ambiguidade como nos tuplos).

Operações com listas

Operação	*Tipo dos argumentos*	*Valor*
l_1 `+` l_2	Listas	A concatenação das listas l_1 e l_2.
l `*` i	Lista e inteiro	A repetição i vezes da lista l.
l`[`i_1`:`i_2`]`	Lista e inteiros	A sub-lista de l entre os índices i_1 e $i_2 - 1$.
`del(`els`)`	Lista e inteiro(s)	Em que els pode ser da forma l`[`i`]` ou l`[`i_1`:`i_2`]`. Remove os elemento(s) especificado(s) da lista l.
e `in` l	Universal e lista	`True` se o elemento e pertence à lista l; `False` em caso contrário.
e `not in` l	Universal e lista	A negação do resultado da operação e `in` l.
`list(`a`)`	Tuplo ou dicionário ou cadeia de caracteres	Transforma o seu argumento numa lista. Se não forem fornecidos argumentos, o seu valor é a lista vazia.
`len(`l`)`	Lista	O número de elementos da lista l.

Mais detalhes das operações possíveis com listas:
https://docs.python.org/3/library/stdtypes.html#typesseq-mutable
(https://docs.python.org/3/library/stdtypes.html#typesseq-mutable)

Listas

Operações com listas, exemplos:

- lst1 = [2, 3, 5, 7]
- lst2 = [0, 1, 1, 2, 3, 5, 8]
- lst1 + lst2
- lst1*5
- lst2[2:5]
- del(lst2[2:5])
- lst2.append(4)
- del list2[-1]
- 8 in lst2
- 8 in lst1
- len(lst1)
- list((8,9,4))
- list('Fundamentos')
- lst1[1] = 'FP'
- lst1[2] = [1, 2, 3]

In [161]:
```
lst1 = [2, 3, 5, 7]
print(lst1, type(lst1))
lst2 = [0, 1, 1, 2, 3, 5, 8]
print(lst2, id(lst2))
lst1[1] = -1
lst1
```

```
[2, 3, 5, 7] <class 'list'>
[0, 1, 1, 2, 3, 5, 8] 4474398472
```

Out[161]:
```
[2, -1, 5, 7]
```

Exemplo listas, acrescentar elementos:

In [176]:
```
lst1 = [1,2,3]
print(id(lst1))
for i in [10, 20]:
    lst1.append(i)

print(lst1)
print(id(lst1))
```

```
4474716296
[1, 2, 3, 10, 20]
4474716296
```

Atribuição em listas: Considerações sobre mutabilidade

In [166]:
```
lst1 = [2, 3, 5, 7]
lst2 = lst1

lst1[2] = 6
lst2[1] = 4

print(lst1)
print(lst2)

lst1 = 10

print(lst1)
print(lst2)
```

```
[2, 4, 6, 7]
[2, 4, 6, 7]
10
[2, 4, 6, 7]
```

Atribuição em listas: Considerações sobre mutabilidade

In [168]:
```
lst1 = [2, 3, 5, 7]
lst2 = list(lst1)
lst3 = lst2[:]

lst2[0] = -1
lst3[1] = -1

print(lst1)
print(lst2)
print(lst3)
```

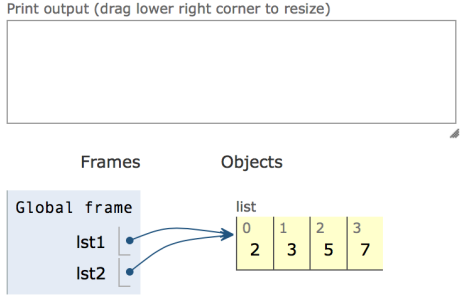
```
[2, 3, 5, 7]
[-1, -1, 5, 7]
[-1, -1, 5, 7]
```

Atribuição em listas: Considerações sobre mutabilidade - Python Tutor

Python 3.6

```
1  lst1 = [2, 3, 5, 7]
2  lst2 = lst1
3  e1 = lst1[1]
4
5  lst1[2] = 6
6  lst1[1] = 4
7
8  print(lst1)
9  print(lst2)
10 print(e1)
11
12 lst1 = 10
13
14 print(lst1)
15 print(lst2)
16 print(e1)
```

Print output (drag lower right corner to resize)



Passagem de parâmetros

- Modo de passagem de parâmetros mais comuns em programação:
  - Por valor - A função recebe o valor do parâmetro concreto e mais nenhuma informação
  - Referência - A função recebe a posição em memória do parâmetro concreto
- Em Python é um pouco diferente:
  - Os parâmetros são passados por *passagem por referência dos objetos* ou *assignment* (valor da referência do objeto).
  - *Assignment* é a operação de ligar (binding) um nome a um objeto.
  - Implicações:
    - Podemos alterar/mudar os objetos que sejam mutáveis.
    - Não podemos fazer *rebinding* da referencia externa, ou seja ligar o nome da variável do ambiente exterior da função a um outro objeto.
- Leituras adicionais:
  - https://jeffknupp.com/blog/2012/11/13/is-python-callbyvalue-or-

callbyreference-neither/ (https://jeffknupp.com/blog/2012/11/13/is-python-callbyvalue-or-callbyreference-neither/)

- https://docs.python.org/3/faq/programming.html#how-do-i-write-a-function-with-output-parameters-call-by-reference (https://docs.python.org/3/faq/programming.html#how-do-i-write-a-function-with-output-parameters-call-by-reference)

Passagem de parâmetros - Exemplo parâmetros imutáveis

In [169]:
```
def func1(a, b):
    print("DENTRO ANTES da troca:", a, b)
    a, b = 'new-value', b + 1            # a and b are local names
    print("DENTRO DEPOIS da troca:", a, b)   # assigned to new objects


a, b = 'old-value', 99
print("FORA ANTES da troca:", a, b)
func1(a, b)
print("FORA DEPOIS da troca:", a,b)
```

```
FORA ANTES da troca: old-value 99
DENTRO ANTES da troca: old-value 99
DENTRO DEPOIS da troca: new-value 100
FORA DEPOIS da troca: old-value 99
```

Passagem de parâmetros - Exemplo parâmetros mutáveis 1

In [170]:
```
def func2(l):
    print("DENTRO ANTES da troca:", l)
    l[0], l[1] = 'new-value', l[1] + 1     # 'a' references a mutable list
    print("DENTRO DEPOIS da troca:", l)

l = ['old-value', 99]
print("FORA ANTES da troca:", l)
func2(l)
print("FORA DEPOIS da troca:", l)
```

```
FORA ANTES da troca: ['old-value', 99]
DENTRO ANTES da troca: ['old-value', 99]
DENTRO DEPOIS da troca: ['new-value', 100]
FORA DEPOIS da troca: ['new-value', 100]
```

Passagem de parâmetros - Exemplo parâmetros mutáveis 2

In [135]:
```
def func3(l):
    print("DENTRO ANTES da troca:", l)
    l[0], l[1] = 'new-value', l[1] + 1     # 'a' references a mutable list
    print("DENTRO DEPOIS da troca:", l)
    l = ['last new-value', l[1] + 1]       # rebinding example
    print("DENTRO DEPOIS da assignment:", l)

l = ['old-value', 99]
print("FORA ANTES da troca:", l)
func3(l)
print("FORA DEPOIS da troca:", l)
```

```
FORA ANTES da troca: ['old-value', 99]
DENTRO ANTES da troca: ['old-value', 99]
DENTRO DEPOIS da troca: ['new-value', 100]
DENTRO DEPOIS da assignment: ['last new-value', 101]
FORA DEPOIS da troca: ['new-value', 100]
```

Passagem de parâmetros - Exemplo *tuplos* mutáveis !?!?

In [178]:
```
def addtoall(t, x):
    for l in t:
        l.append(x)


t = ([],[])
addtoall(t, 2)
addtoall(t, 17)
addtoall(t, 4)
print(t)
```

```
([2, 17, 4], [2, 17, 4])
```

Exemplo: Verificar IMEI válido

- O IMEI (International Mobile Equipment Identity) é um número de 15 dígitos, geralmente exclusivo, para identificar telefones móveis (marcar *#06# para o obter). O dígito mais à direita é um dígito de verificação ou *checksum*.

- O algoritmo Luhn ou a fórmula Luhn, também conhecido como algoritmo *módulo 10*, é uma fórmula de *checksum* simples usada para validar uma variedade de números de identificação, como números de cartão de crédito, números IMEI, etc.

- O algoritmo de Luhn verifica um número em relação ao seu dígito de verificação. O número deve passar no seguinte teste:

    - A partir do dígito mais à direita, que é o dígito de verificação, e movendo para a esquerda, dobrar o valor de cada segundo dígito. Se o resultado dessa operação de duplicação for maior que 9 (por exemplo, 8 × 2 = 16), adicionar os dígitos do número (por exemplo, 16: 1 + 6 = 7, 18: 1 + 8 = 9) ou, alternativamente, , o mesmo resultado pode ser encontrado ao subtrair 9 (por exemplo, 16: 16 - 9 = 7, 18: 18 - 9 = 9).

    - Calcular a soma de todos os dígitos, incluindo o dígito de verificação

    - Se o módulo 10 total é igual a 0 então o número é **válido** de acordo com a fórmula de Luhn; senão é **não válido**.

Exemplo: Verificar IMEI válido - Proposta solução 1:

In [179]:
```
def is_valid_imei(imei):
    """
    checks if an imei is valid.
    """

    if (not isinstance(imei, str)) or len(imei) != 15:
        raise ValueError("Doesn't look like a serial number!")

    imei = list(imei)
    for i in range(len(imei)):
        imei[i] = int(imei[i])

    for i in range(len(imei) - 2, -1, -2):
        imei[i] = imei[i] * 2

    print(imei)

    for i in range(len(imei)):
        if imei[i] >= 10:
            imei[i] -= 9

    print(imei)

    acc = 0
    for i in range(len(imei)):
        acc = acc + imei[i]

    return acc % 10 == 0

is_valid_imei("353270079684223")
```

```
[3, 10, 3, 4, 7, 0, 0, 14, 9, 12, 8, 8, 2, 4, 3]
[3, 1, 3, 4, 7, 0, 0, 5, 9, 3, 8, 8, 2, 4, 3]
```

Out[179]:
```
True
```

Exemplo: Verificar IMEI válido - Proposta solução 2:

In [38]:
```
#Yet another solution.

def is_valid_imei2(imei):
    """
    checks if an imei is valid.

    """
    if (not isinstance(imei, str)) or len(imei) != 15:
        raise ValueError("Doesn't look like a serial number!")

    imei = list(imei)
    acc = 0
    factor = 1
    for i in range(len(imei) - 1, -1, -1):
        double = int(imei[i]) * factor
        if double > 9:
            acc = acc + (double - 9)
        else:
            acc = acc + double

        if factor == 2:
            factor = 1
        else:
            factor = 2

    return acc % 10 == 0

is_valid_imei2("353270079684223")
```

Out[38]:
```
True
```

Lists comprehensions (avançado)

- O Python suporta um conceito chamado *list comprehensions* que pode ser usado para construir listas de uma maneira muito natural e fácil, parecido como na matemática.
- As *lists comprehensions* é uma das componentes de Python relacionados com **programação funcional** que iremos a ver em mais pormenor nas próximas semanas.
- A seguir estão formas comuns de descrever vetores (ou tuplos ou listas) em matemática:
  - $S = \{x^2 : x \text{ in } \{0 \ldots 9\}\}$
  - $V = (1, 2, 4, 8, \ldots, 2^{12})$
  - $M = \{x \mid x \text{ in } S \text{ and } x \text{ even}\}$

Lists comprehensions (avançado)

- Em Python, é possível escrever essas expressões quase exatamente como um matemático faria, sem precisar de se lembrar de nenhuma sintaxe críptica especial.
- É assim que se faz em Python:

```
>>> S = [x**2 for x in range(10)]
>>> V = [2**i for i in range(13)]
>>> M = [x for x in S if x % 2 == 0]
```

In [42]:
```
S = [x**2 for x in range(10)]
print(S)
V = [2**i for i in range(13)]
print(V)
[x for x in S if x % 2 == 0]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
```

Out[42]:
```
[0, 4, 16, 36, 64]
```

Exemplo - Crivo de Eratóstenes

Problema: Dado um número `n`, imprima todos os primos menores ou iguais a `n`. Por exemplo, se `n = 20`, a saída deve ser `2, 3, 5, 7, 11, 13, 17, 19`.

A crivo de Eratóstenes é uma das formas mais eficientes de encontrar todos os primos menores que `n` quando `n` for menor que 10 milhões. Algoritmo:

- Crie uma lista de inteiros consecutivos de 2 a n: `lista = [2, 3, 4, ..., n]`.
- Seleciona o primeiro elemento da lista, `p = 2`.
- Enquanto `p` não for maior que $\sqrt{n}$:
  - (a) removem-se da lista todos os múltiplos de p;
  - (b) passa-se ao número seguinte na lista.
- No final do algoritmo, a lista apenas contém números primos.

Exemplo - Crivo de Eratóstenes, Proposta 1

In [57]:
```
from math import sqrt

def crivo1(n):

    lista = list(range(2, n+1))

    i = 0
    while lista[i] <= sqrt(n):
        p = lista[i]
        j = i + 1
        while j < len(lista):
            if lista[j] % p == 0:
                del lista[j]
            else:
                j += 1

        i = i + 1

    return lista

crivo1(50)
# Question1: Sera que podemos começar com uma lista menor? Reparem que o único par
 primo é o 2
# Question2: Porque não podemos utilizar for?
# Question3: Sera que podemos abstrair em uma função (mais eficiente) a elimínção
 dos múltiplos
```

Out[57]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Exemplo - Crivo de Eratóstenes, Proposta 2

In [58]:
```
def crivo2(n):

    # redizimos a lista inicial à metade tirando os pares
    lista = [2] + list(range(3, n+1, 2))

    i = 0
    limite = sqrt(n)
    while lista[i] <= limite:
        # abstraimos a eliminação de multiplos numa função que consiga utilizar fo
r
        elimina_multiplos(lista, i)
        i = i + 1

    return lista


def elimina_multiplos(lista, index):
    p = lista[index]
    fim = len(lista) -1
    for i in range(fim, index, -1):
        if lista[i] % p == 0:
            del lista[i]

crivo2(50)
# Question 1: Sera que podemos manter o tamanho da lista fixo e utilizar fors?
```

Out[58]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Exemplo - Crivo de Eratóstenes, Proposta 3

In [46]:
```
def crivo3(n):

    # lista de bools correspondente a inteiros de 0..n indicando se é ou não primo
    lista = [True]*(n+1)
    lista[0], lista[1] = False, False

    # equivalente ao primeiro while anterior
    for i in range(2, int(sqrt(n)) + 1):
        # colocamos a False as possições multiplas de i
        for j in range(i+i, n+1, i):
            lista[j] = False

    return [i for i in range(n+1) if lista[i]]

crivo3(50)
```

Out[46]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Exemplo - Crivo de Eratóstenes, Eficiência

In [182]:
```
n = 300000
%time crivo1(n)
print()

%time crivo2(n)
print()

%time crivo3(n)
print()
```

```
CPU times: user 6.15 s, sys: 8.88 ms, total: 6.15 s
Wall time: 6.15 s

CPU times: user 1.11 s, sys: 1.64 ms, total: 1.11 s
Wall time: 1.11 s

CPU times: user 94.1 ms, sys: 1.32 ms, total: 95.4 ms
Wall time: 96.7 ms
```

In [183]:
```
# %load_ext memory_profiler

%memit crivo1(n)
print()

%memit crivo2(n)
print()

%memit crivo3(n)
print()
```

```
peak memory: 47.74 MiB, increment: 7.49 MiB

peak memory: 40.74 MiB, increment: 2.43 MiB

peak memory: 39.22 MiB, increment: 0.81 MiB
```

Algoritmos de procura

- A procura de um elemento em uma **lista** é uma das operações mais comuns sobre listas
- O objetivo do processo de procura em uma lista `l` é descobrir se o valor `x` está na lista e em que possição.
- Existem múltiplos algoritmos de procura (alguns mais eficientes e outros menos).
- Hoje vamos ver:
  - Procura sequencial ou linear
  - Procura binária

Algoritmos de procura - Procura sequencial

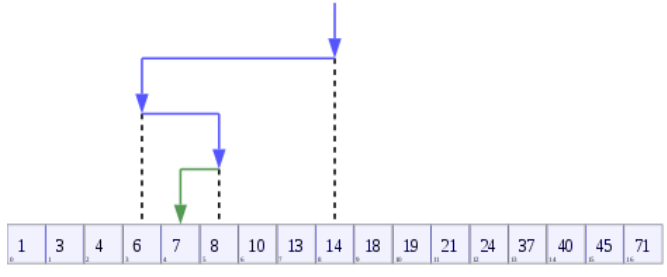
In [63]:
```
def linearsearch(l, x):
    for i in range(len(l)):
        if l[i] == x:
            return i
    return -1

linearsearch([1,2,3], 2)
```

Out[63]: 1

- O numero de comparações depende da possição onde se encontrar o elemento, pode ir de `1` até `n` se o elemento não se encontrar na lista
- Será que conseguimos fazer melhor?

Algoritmos de procura - Procura binária

- Podemos fazer melhor se a lista estiver ordenada!!



In [31]:
```
def binsearch(l, x):
    left = 0
    right = len(l) - 1

    while left <= right:
        mid = left + (right - left)//2
        if x == l[mid]:
            return mid
        elif x > l[mid]:
            left = mid + 1
        else:
            right = mid - 1
    return -1

binsearch([1,2,3], 3)
```

Out[31]:
```
2
```

Algoritmos de ordenação

- Isto não significa que sempre seja melhor ordenar e procurar depois.
- Em geral, a ordenação têm um custo superior que a procura linear, e manter uma lista ordenada também é custoso.
- No entanto, se o número de procuras for muito superior ao número de alterações na lista, compensa ordenar e utilizar a pesquisa binária.
- Existem vários algoritmos de ordenação (https://en.wikipedia.org/wiki/Sorting_algorithm) e, em Python, temos as funções pré-definidas `sorted` e a função `sort` sobre listas, que implementa um desses algoritmos de ordenação chamado *Timsort*.

```
>>> l = [1,8,21,4,1,8,9]
>>> sorted(l)
[1, 1, 4, 8, 8, 9, 21]
>>> l
[1, 8, 21, 4, 1, 8, 9]
>>> l.sort()
>>> l
```

```
[1, 1, 4, 8, 8, 9, 21]
>>>
```

In [54]:

```
l = [1,8,21,4,1,8,9]
l = sorted(l)
print(l)
```

```
[1, 1, 4, 8, 8, 9, 21]
```

Algoritmos de ordenação - Bubble sort

In [86]:
```
from random import shuffle
nums = list(range(10000))
shuffle(nums)


def bubblesort(l):
    changed = True
    size = len(l) - 1
    while changed:
        changed = False
        for i in range(size): #maiores para o fim da lista
            if l[i] > l[i+1]:
                l[i], l[i+1] = l[i+1], l[i]
                changed = True
        size = size -1

nums1=nums[:]
%time bubblesort(nums1)
print(nums1 == sorted(nums1))
```

```
CPU times: user 8.19 s, sys: 15.4 ms, total: 8.21 s
Wall time: 8.23 s
True
```

Algoritmos de ordenação - Shell sort

In [87]:
```
def bubblesort(l, step = 1):
    changed = True
    size = len(l) - step
    while changed:
        changed = False
        for i in range(size): #maiores para o fim da lista
            if l[i] > l[i+step]:
                l[i], l[i+step] = l[i+step], l[i]
                changed = True
        size = size -1

def shellsort(l):
    step = len(l)//2
    while step != 0:
        bubblesort(l, step)
        step = step//2


nums = list(range(10000))
shuffle(nums)

nums1=nums[:]
%time bubblesort(nums1)
print(nums1 == sorted(nums1))

nums2=nums[:]
%time shellsort(nums2)
print(nums2 == sorted(nums2))
```

```
CPU times: user 8.1 s, sys: 14.8 ms, total: 8.11 s
Wall time: 8.13 s
True
CPU times: user 146 ms, sys: 212 µs, total: 146 ms
Wall time: 146 ms
True
```

Algoritmos de ordenação - Selection sort

In [89]:
```
def selectionsort(lista):

    for i in range(len(lista)):
        minimum = i
        for j in range(i+1, len(lista)):
            if lista[j] < lista[minimum]:
                minimum = j
        lista[i], lista[minimum] = lista[minimum], lista[i]

nums3=nums[:]
%time selectsort(nums3)
print(nums3 == sorted(nums3))
```

```
CPU times: user 3.62 s, sys: 13.3 ms, total: 3.63 s
Wall time: 3.65 s
True
```

Algoritmos de ordenação - Insertion sort

In [90]:
```
def insertionsort(l):
    for i in range(1, len(l)):
        x = l[i]
        j = i - 1
        while j >= 0 and x < l[j]:
            l[j+1] = l[j]
            j = j - 1
        l[j+1] = x

nums4=nums[:]
%time selectsort(nums4)
print(nums4 == sorted(nums4))
```

```
CPU times: user 3.66 s, sys: 17 ms, total: 3.68 s
Wall time: 3.69 s
True
```