

# **Programação II**

**+**

# **Estruturas de Dados para**

# **Bioinformática**

Hugo Pacheco

DCC/FCUP

23/24

Iteração

# Fluxo de programas

- Python é uma linguagem imperativa
- Programa definido como sequência de instruções
- Tipicamente 1 linha = 1 instrução, mas nem sempre

```
print(3)  
print(4)
```

```
print(3) ; print(4)
```

- Fluxo do programa = instrução a instrução, de cima para baixo

# Um programa Turtle

- módulo turtle

```
import turtle
```

- configuração da janela

```
window = turtle.Screen()  
window.bgcolor("lightgreen")  
window.title("Hello, Alex!")
```

- configuração da tartaruga

```
alex = turtle.Turtle()  
alex.color("blue")  
alex.pensize(3)
```

- movimento da tartaruga

```
alex.forward(50)  
alex.left(120)  
alex.forward(50)
```

- fazer janela esperar

```
#window.mainloop()
```

# Um programa Turtle

- módulo *turtle*

```
import turtle
```

- objectos (têm estado interno “escondido”)

*window : Screen*

*alex : Turtle*

- Métodos (alteram o estado interno)

*objeto.método(args)*

```
window = turtle.Screen()  
window.bgcolor("lightgreen")  
window.title("Hello, Alex!")
```

```
alex = turtle.Turtle()  
alex.color("blue")  
alex.pensize(3)
```

```
alex.forward(50)  
alex.left(120)  
alex.forward(50)
```

```
#window.mainloop()
```

# Iteração (ciclo for)

- Um dos elementos principais em programação é a repetição de instruções = iteração
- Uma das formas de iteração mais simples é o ciclo **for** (indentação importante)

```
for friend in ["Joe", "Zoe", "Zuki", "Thandi", "Paris"]:  
    invite = "Hi " + friend + ". Please come to my party!"  
    print(invite)
```

```
for x in "banana":  
    print(x)
```

```
for x in range(6):  
    print(x)
```

```
for x in ["red", "big", "tasty"]:  
    for y in ["apple", "banana", "cherry"]:  
        print(x, y)
```

# Iteração (ciclo for + turtle)

- Desenhar um quadrado

```
alex.forward(100)
alex.left(90)
alex.forward(100)
alex.left(90)
alex.forward(100)
alex.left(90)
alex.forward(100)
alex.left(90)
```

- Repetição de padrões

```
for _ in range(4):
    alex.forward(100)
    alex.left(90)
```

# Booleanos

- Um **bool** é **True** ou **False**

```
print(bool("Hello"))  
print(bool(15))  
print(bool(""))  
print(bool(0))
```

- casts

```
print(10 == 9)  
print(10 != 9)  
print(10 > 9)  
print(9 >= 9)  
print(10 < 9)  
print(10 <= 9)
```

- comparação

- lógica booleana

```
print(x < 5 and x < 10)  
print(x < 5 or x < 4)  
print(not(x < 5 and x < 10))
```



# Condicionais

- Um dos elementos principais em programação é o comportamento por casos = condicionais

```
if x % 2 == 0: print(x, "is even")
else: print(x, "is odd")
```

```
print(x, "is even") if x % 2 else print(x, "is odd")
```

```
if x > y: print("x is greater than y")
elif x == y: print("x and y are equal")
else: print("y is greater than x")
```

```
if 0 < x:
    if x < 10:
        print("x is a positive single digit.")
```

```
if 0 < x and x < 10:
    print("x is a positive single digit.")
```

# Condicionais aninhados

- Python não suporta sintaxe “switch” para definir vários casos
- Pode ser simulada com dicionários

**C**

```
switch (x)
{
    case 1: print("A"); break;
    case 2: print("B"); break;
    default: print("C");
}
```

**Python**

```
if x==1: print("A")
elif x==2: print("B")
else: print("C")
```

```
d = {1:print("A"), 2:print("B")}
if x in d: d[x] else print("C")
```

# Ciclo for

- Permite repetir instruções um número fixo de vezes
- Percorrer um **iterador**, i.e., uma sequência de elementos
- Pode ser uma string, um range numérico, uma lista, etc
- E.g., somar uma lista de números inteiros

```
numbers = [5, 6, 32, 21, 9]
running_total = 0
for number in numbers:
    running_total += number
print(running_total)
```

# Ciclo while

- Permite repetir instruções um número indefinido de vezes, controlado dinamicamente pelo próprio ciclo
- E.g., somar uma lista de números inteiros (inicialização, condição do ciclo, atualização)

```
numbers = [5, 6, 32, 21, 9]
running_total = 0
i = 0;
while (i < len(numbers)) :
    running_total += numbers[i]
    i+=1
print(running_total)
```

# Ciclo + condicional

- Um ciclo pode ter um corpo condicional

```
xs = [12, 16, 17, 24, 29]
```

```
for x in xs:  
    if x % 2 == 0: print(x)
```

```
for x in xs:  
    if x % 2 == 0: print(x)  
    else: break;
```

```
i=0  
while (i < len(xs) and xs[i] % 2 == 0):  
    print(xs[i]); i+=1
```

# Ciclo while (Collatz)

- Função de Collatz

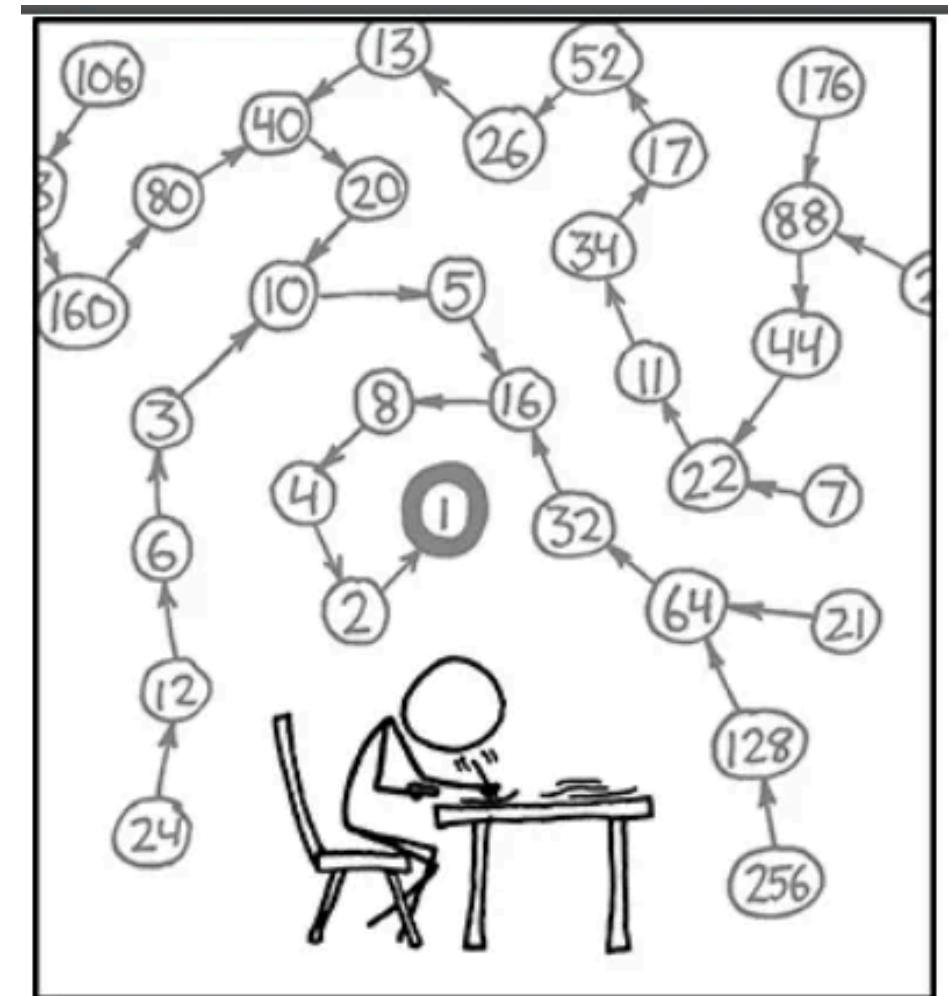
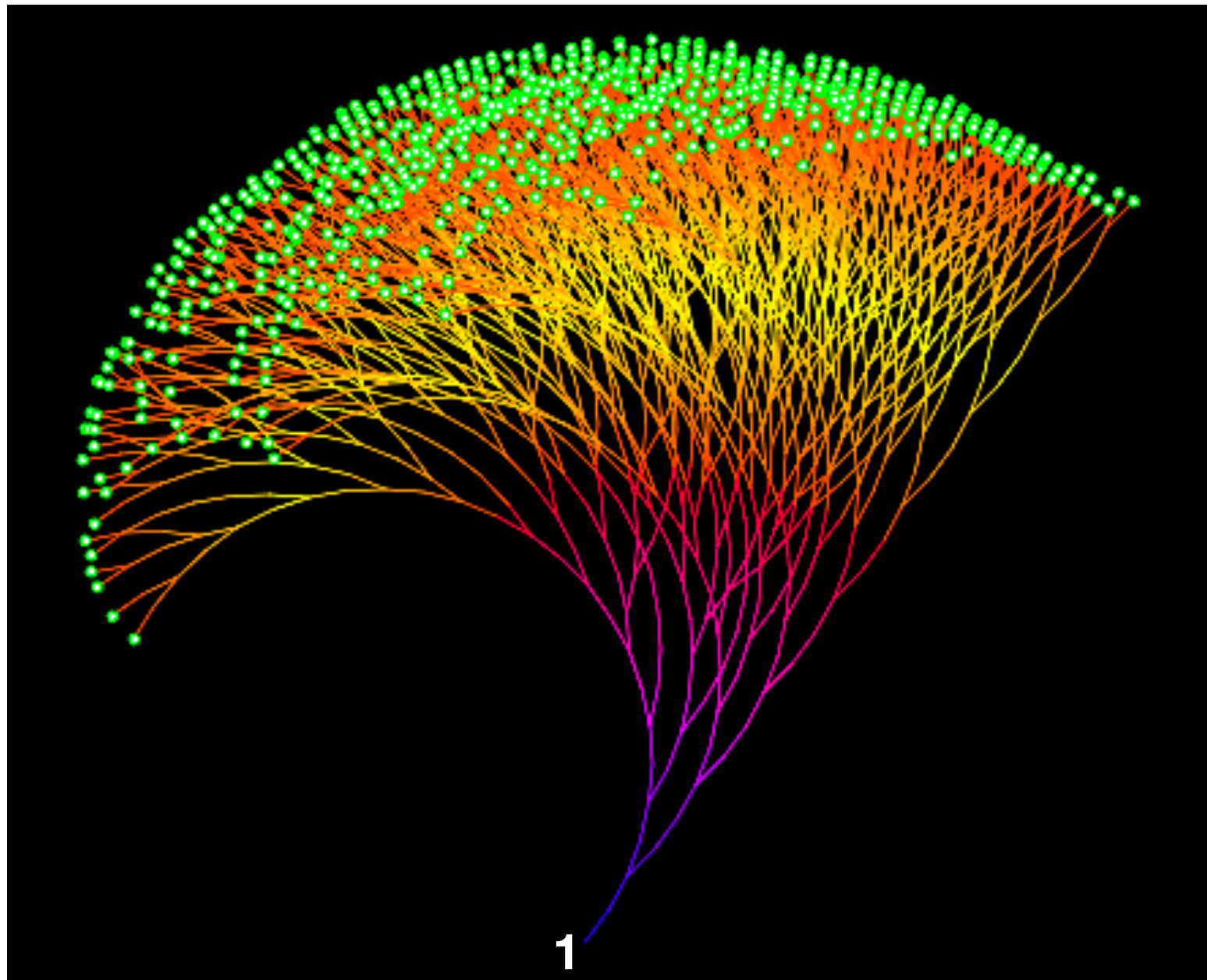
$$f(n) = \begin{cases} n/2, & n \text{ par} \\ 3n + 1, & n \text{ ímpar} \end{cases}$$

- Calcular sequência de Collatz para  $n > 0$  enquanto  $n \neq 1$

```
while n != 1:
    if n%2 == 0: n = n//2
    else: n = 3*n+1
    print(n)
```

- Nota: não sabemos se esta função termina para todo o  $n > 0$ !

# Ciclo while (Collatz)



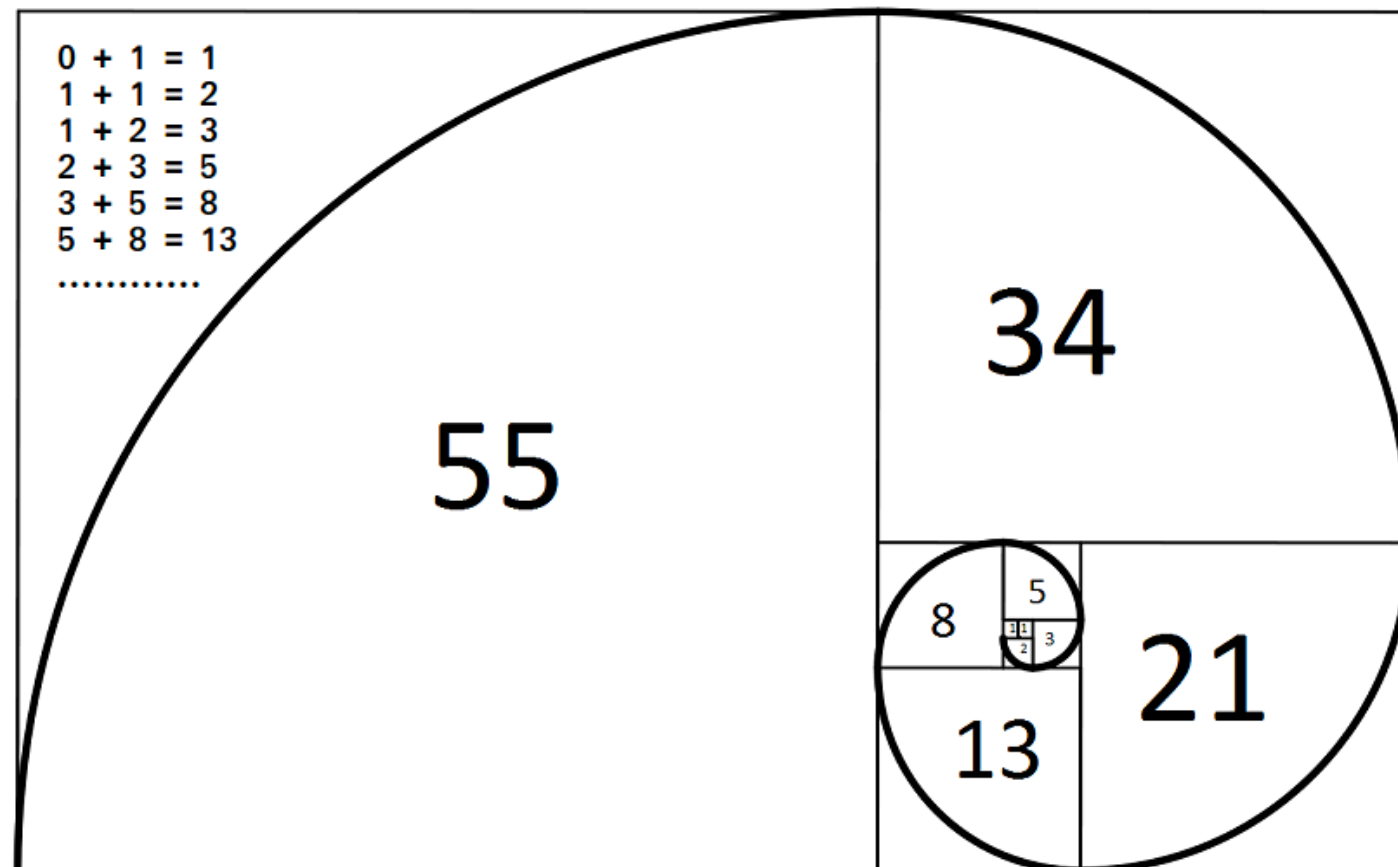
THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

# Ciclo while (Fibonnaci)

- Função de Fibonnaci

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & n > 1 \end{cases}$$

- Desenhar sequência de Fibonnaci





# Ciclo while (fib + turtle)

```
import turtle
import math

window = turtle.Screen(); alex = turtle.Turtle()

prev = 0 # fib(n-1)
fib = 1 # fib(n)

alex.right(90)
while True:
    arc = 2 * math.sin(math.pi / 360) * (fib * 10)
    for j in range(90):
        alex.forward(arc)
        alex.left(1)
    fib, prev = fib + prev, fib
```

# Recursividade

# Funções (fluxo)

- Definição de funções não altera fluxo do programa
- Pode definir-se funções no meio do código, mas não é recomendado
- Função apenas é executada quando chamada, fluxo salta para a 1ª linha da função, e retorna ao ponto onde estava pós saída da função
- Função pode utilizar parâmetros globais
- Função tem parâmetros e variáveis locais, destruídos à saída

```
g = 9.8
def y(v0, t):
    return v0 * t - 1 / 2 * g * t * t
v0=1
print("t=0:", y(v0, 0))
print("t=2:", y(v0, 2))
```

# Recursividade

- Funções podem invocar-se a si próprias
- Cada invocação é tratada como uma nova função
- E.g., somar uma lista de números inteiros

```
numbers = [5, 6, 32, 21, 9]
def sum(xs):
    if xs == []: return 0
    else: return xs[0] + sum(xs[1:])

print(sum(numbers))
```

# Recursividade

- E.g., função de Fibonnaci

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & n > 1 \end{cases}$$

```
def fib(n):  
    if n == 0: return 0  
    elif n == 1: return 1  
    else: return fib(n-1) + fib(n-2)
```

- E.g., função factorial

$$n! = \begin{cases} 1, & n = 0 \\ n(n-1)!, & n > 0 \end{cases}$$

```
def fact(n):  
    if n == 0: return 1  
    else: return n * fact(n-1)
```

# Funções ou Procedimentos

- Funções matemáticas são puras, i.e., só dependem dos seus argumentos e não têm efeitos laterais (à la programação funcional, **preferível**)
- Funções Python podem não ser puras (chamadas de procedimentos), por exemplo, quando nem têm valor de retorno (só justificável por questões de eficiência)

```
def f(x):  
    y = x**2  
    print(y)  
print(f(2))
```

```
y = 0  
def f(x):  
    global y  
    y = x**2  
print(y)
```

# Recursividade (Collatz)

- Imprimir a sequência de Collatz

```
def collatz(n):  
    if (n == 1):  
        return None  
    elif (n % 2 == 0):  
        print(n//2)  
        collatz(n // 2)  
    else:  
        print(3*n+1)  
        collatz(3 * n + 1)  
  
collatz(10)
```

# Funções (turtle)

- Desenhar uma espiral de quadrados multicolores

```
import turtle

def draw_multicolor_square(animal, size):
    for color in ["red", "purple", "hotpink", "blue"]:
        animal.color(color)
        animal.forward(size)
        animal.left(90)

window = turtle.Screen(); alex = turtle.Turtle()

size = 20
while True:
    draw_multicolor_square(alex, size)
    size += 10
    alex.forward(10)
    alex.right(18)

window.mainloop()
```



Tipos base

# Tipos compostos

- Valores de tipos primitivos (*int*, *float*, *bool*, etc) são indivisíveis

```
math.sqrt(5 + 3)
```

- Valores de tipos compostos (*str*, *list*, *tuple*, etc) podem ser vistos como um todo...

```
our_string = "Hello, World!"  
print(our_string.upper())  
print(len(our_string))
```

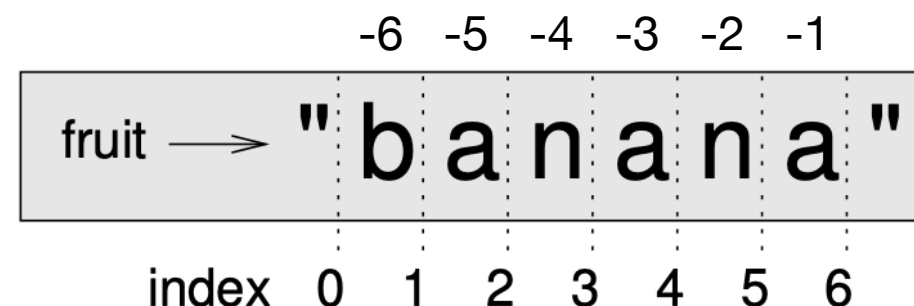
- ...ou como coleções de valores mais pequenos
- Tipos compostos são **objetos** que podem ter atributos, métodos e estado interno. Mais importante, **pode** haver efeitos secundários!



# Strings

- Strings são sequências de caracteres, em que cada caracter é por sua vez uma string unitária
- Podemos aceder a elementos utilizando indexação, com índices [0 ..len-1] ou [-1..-len].

```
fruit = "banana"  
letter = fruit[1]  
print(letter.upper())
```



- O mesmo método de indexação funciona para listas ou tuplos

# Listas e tuplos

- Listas são sequências de tamanho variável
- Tuplos são sequências de tamanho fixo
- Ao contrário de outras linguagens, sequências podem ter elementos de diferentes tipos (tipos dinâmicos)
- É possível converter um no outro

```
[ ]  
[1, 2]  
['a', 'b', 'c']
```

```
()  
(1, 2)  
( 'a', 'b', 'c' )
```

```
[1, 3.5, True]  
(1, 'a', False)
```

```
print(tuple([1, 'a']))  
print(list((1, 'a')))
```

# Coleções (indexação)

```
fruit = "banana"  
fruits = list(enumerate(fruit))  
triple = (1, fruit, False)
```

- Indexação por índice positivo (do início para o fim)

```
print(fruit[1])  
print(fruits[1])  
print(fruits[len(fruit)-1])  
print(triple[2])
```

- Indexação por índice negativo (do fim para o início)

```
print(fruit[-1] == fruit[5])  
print(fruit[-1] == fruit[len(chars)-1])  
print(triple[-2] == triple[1])
```

# Coleções (*slices*)

- É também possível obter sub-coleções
  - `txt[i:j]` substring entre índices  $i$  e  $j-1$  inclusive
  - `txt[i:]` substring desde o índice  $i$  até ao final
  - `txt[:j]` substring desde o início até ao índice  $j-1$  inclusive

```
phrase = "Pirates of the Caribbean"
print(phrase[0:5])
friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki",
           "Thandi", "Paris"]
print(friends[2:])
triple = (1, "banana", True)
print(triple[1:])
```

# Tuplos (*unpacking*)

- É possível abrir um tuplo atribuindo uma variável a cada elemento

```
bob = ("Bob", 19, "CS")  
(name, age, studies) = bob  
print(name, age, studies)
```

- e usar essa sintaxe para trocar o valor de duas variáveis

```
a = "Gin"  
b = "tonic"  
(a, b) = (b, a)  
print(a, b)
```

# Tuplos (*unpacking*)

- Também chamado de *pattern matching*
- O caracter \* permite selecionar vários sub-sequências (como uma lista)

```
a, *bs = [1, 2, 3, 4]
a, *bs, c = (1, 2, 3, 4)
*bs, *cs = [1, 2, 3, 4] # erro
```

- Podemos compor padrões aninhados

```
p= ("a", (2, 3.5), [3, 4, 5, 6])
(p1, (p21, p22), [p31, *p32, p33])=p
```



# Coleções (iteração)

- É possível iterar pelos elementos de uma sequência
- Existem outros tipos de coleções, e.g. o resultado da função *enumerate*, e seguem o mesmo padrão

```
word = "banana"  
lst = [1, 2, 3.5, True]  
triple = (1, word, False)
```

```
for letter in word: print(letter)  
for i, char in enumerate(word): print(i, letter)  
for char in lst: print(char)  
for el in triple: print(el)
```

# Coleções (comparação)

- É possível comparar sequências do mesmo tipo, comparando elemento a elemento
  - caracteres são comparados por ordem lexicográfica (ou seja, pelo seu código ASCII)

```
>>> ord('a')
97
>>> ord('z')
122
>>> ord('Z')
90
>>> chr(104)
'h'
>>> chr(97)
'a'
```

```
word = "banana"
words = list(word)
list1 = [1, 2, 3]
list2 = [0, 5, 6]
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 3, 4)

print(word == "banana")
print("zebra" < "banana")
print("Zebra" < "banana")
print("zebra" < "Banana")
print(word == words)
print(list2 < list1)
print(words < list1) # erro
print(tuple1 < tuple2)
```

# Coleções (update)

- Strings e tuplos são **imutáveis**; não podem ser alterados, apenas ser copiados

```
greeting = "Jello, world!"  
tuple=(1,4,6)
```

```
greeting[0] = 'H' #erro  
tuplo[0] = 2 #erro
```

```
new_greeting = "H" + greeting[1:]  
new_tuple = (2,tuple[1],tuple[2])
```

- Listas são **mutáveis**, podem ser alteradas “in-place”

```
xs = [1,2,3,4,5,6]
```

```
xs[0] = 'a' #update first  
xs[-1] = 'd' #update last  
xs[1:3] = ['b','c'] #update multiple  
xs[3:5] = [] #delete indexes 3 and 4 (não recomendado)  
xs[0:1] = [True,False] # insert at head (não recomendado)
```

# Listas (update)

- Outras modificações “in-place” de listas

```
mylist = [5, 27, 3]

# acrescentar elemento no fim
mylist.append(12)
# acrescentar lista no fim
mylist.extend([5, 9, 5, 11])
# inserir elemento numa posição
mylist.insert(1, 12)
# remover primeira ocorrência de elemento
mylist.remove(12)
# remover elemento por índice
del mylist[0]
```

- Tipicamente (consultar documentação):
  - Funções ou métodos que produzem novas listas retornam listas
  - Métodos que alteram a lista “in-place” não retornam listas

# Listas (cópia)

- **Aliasing:** duas variáveis podem referir-se ao mesmo objeto do tipo lista

```
a = [1, 2, 3]
b = a
b[0]=0
print(a,b)
```

- Logo alterar uma das variáveis altera ambas
- Em certos casos pode ser útil modificar uma lista, mas garantindo que a lista original é preservada

```
a = [1, 2, 3]
b = a[:]
#ou b = list(a)
#ou b = a.copy()
b[0]=0
print(a,b)
```

- **Cópia:** cria uma nova lista

# Objetos (igualdade)

- Em Python, por defeito, dois objetos são iguais se apontarem para a mesma região de memória
- É a chamada **igualdade superficial**, que corresponde ao operador *is*
- Cuidado com o operador `==`, comportamento depende da classe!



```
>>> import turtle
>>> alex = turtle.Turtle()
>>> john = turtle.Turtle()
>>> alex==john
False
```

```
>>> l1 = [1,2,3]
>>> l2 = [1,2,3]
>>> l1 is l2
False
>>> l1 == l2
True
```

# Coleções (concatenação)

- É possível concatenar duas (ou mais) coleções do mesmo tipo numa só coleção
- Cria sempre um novo valor (também para listas)

```
print("hello"+"world")
xs = [1,2,3]
ys = ['a','b','c']
zs = xs + ys
print(xs,ys,zs)
print((1,'a',True) + (2,'b',False))
print(list("hello")+ [1,2,3])
print(list((1,'a',True)) + [1,2,3])
print(tuple(zs) + (1,'a',True))
```

# Coleções (repetição)

- É possível repetir (concatenar consigo própria) uma coleção um dado número de vezes
- Cria sempre um novo valor (também para listas)

```
print("hello"*2)
print([1,2]*2)
print((1,2)*3)
```



# Coleções (pertença)

- É possível testar se um elemento pertence a uma coleção; para strings também é possível testar inclusão de substrings

```
word = "banana"
xs = [1, 2, 3, 4, 5, 6]
tupl = (1, 2, 3.5)

print('a' in word)
print('ana' in word)
print(1 in xs)
print(3.5 in tupl)
```

# Coleções (pesquisa)

- É possível procurar o índice de um elemento numa coleção

```
a = "Hello, World!"  
xs = [1, 2, 3]  
t = (1, 2, 3)
```

```
print(a.find('e'))  
xs = [1, 2, 3]  
print(xs.index(2))  
t = (1, 2, 3)  
print(t.index(3))
```

- Ou, para strings, procurar o índice (posição inicial) de uma substring

```
print(a.index('ll'))
```

# Strings (outras operações)

- *strip*: remove espaços
- *replace*: substituir ocorrências
- *split*: divide por separador numa lista de palavras
- *splitlines*: divide por linhas
- *startswith/endswith*: testa se string começa com prefixo/acaba com sufixo
- *join*: concatena lista de strings com separador
- documentação completa

```
a = " Hello, World! "  
b = "Hello\nWorld"
```

```
print(a.strip())
```

```
print(a.replace("He", "J"))
```

```
print(a.split(", "))
```

```
print(a.startswith(' He '))
```

```
print(a.endswith('! '))
```

```
ls = b.splitlines()  
print(ls)
```

```
print(" ".join(ls))
```

# Strings (funções)

- remove caracteres selecionados

```
def remove_chars(chars, phrase) :  
    string_sans_chars = ""  
    for letter in phrase:  
        if letter not in chars:  
            string_sans_chars += letter  
    return string_sans_chars  
  
print(remove_chars("aeiou", "Hello World"))
```

- Nota: é fácil de adaptar para outras coleções

# Listas (outras operações)

- *reverse*: inverte a ordem

```
xs = [1, 2, 3]
xs.reverse()
print(xs)
```

- *clear*: apaga todos os elementos

```
ys = [1, 2, 3]
ys.clear()
print(ys)
```

- *zip*: junta duas listas

```
zs = ['x', 'y', 'z']
ws = [1, 2, 3, 4, 5]
print(list(zip(zs, ws)))
```

- documentação completa

# Coleções (ordem superior)

- Existem funções ditas de ordem superior (porque recebem funções como argumentos) que codificam padrões típicos sobre coleções; o resultado é geralmente uma coleção especial
  - *map*: aplicar uma função a cada elemento da coleção
  - *filter*: selecionar apenas alguns elementos da coleção
  - *sum*: soma todos elementos de uma coleção numérica

```
l = [1, 2, 4.5, True]
```

```
s = "hello123"
```

```
t = (1, 2, 4.5, True)
```

```
def succ(i):
```

```
    return i+1
```

```
def alpha(c):
```

```
    return c.isalpha()
```

```
co1 = map(succ, l)
```

```
print(list(co1))
```

```
co2 = filter(alpha, s)
```

```
print(list(co2))
```

```
print(sum(t))
```

# Coleções (ordenação)

- É possível ordenar os elementos de uma coleção com funções de ordem superior
  - o resultado é uma lista, pela ordem original ou invertida; recebe como argumento opcional uma função que serve como chave para a comparação
- É possível ordenar uma lista “in-place” (método *sort*)

```
word = "banana"
lst = [1, 2, 3.5, True]
triple1 = (1, 3.5, True)
triple2 = (word, lst, (4, 5))

print(sorted(word))
print(sorted(lst, reverse=True))
print(sorted(triple1))
print(sorted(triple2, key=len))

lst.sort(reverse=True); print(lst)
```

# Strings (padrões)

- Uma forma poderosa de encontrar padrões em strings é utilizando as chamadas **expressões regulares**
- Uma expressão regular é uma string com caracteres especiais. Por vezes difícil de ler
- Consultar documentação

.	Qualquer caracter	A*	Zero ou mais repetições de A
^	Início da string	A+	Uma ou mais repetição de A
\$	Fim da string	A?	Zero ou uma ocorrência de A
[ab]	Conjuntos de caracteres	A   B	Alternativa
[a-z]		(A)	Parêntesis
a	Caracter fixo		



# Strings (padrões)

- Podemos pesquisar um padrão (parcialmente ou por completo) numa string

```
import re
```

```
r = re.compile("[o|O].[a|A]")
```

```
print(r.search("OlA"))
```

```
# <re.Match object; span=(0, 3), match='OlA'>
```

```
m = r.search("socapa")
```

```
print(m.start(), m.end(), m.group())
```

```
# 1 4 oca
```

```
r = re.compile("abc(.+)")
```


```
print(r.fullmatch("abcdef"))
```

```
# <re.Match object; span=(0, 6), match='abcdef'>
```

```
print(r.fullmatch("abc"))
```

```
# None
```

# Strings (padrões)

- Podemos pesquisar todas as ocorrências de um padrão
- Módulo **regex** oferece mais funcionalidade 

```
import re
```

```
r = re.compile("[A-z].[A-z]")
print(r.findall("Eu gosto de Python"))
# ['u g', 'ost', 'o d', 'e P', 'yth']
```

```
import regex as re
```

```
r = re.compile("[A-z].[A-z]")
print(r.findall("Eu gosto de Python", overlapped=True))
# ['u g', 'gos', 'ost', 'sto', 'o d', 'e P', 'Pyt', 'yth', 'tho', 'hon']
```

```
for m in r.finditer("asdadasd") :
    print(m)
# <regex.Match object; span=(0, 3), match='asd'>
# <regex.Match object; span=(3, 6), match='asd'>
# <regex.Match object; span=(6, 9), match='asd'>
```