

Programação II + Estruturas de Dados para Bioinformática

Strings, listas e tuplos

Hugo Pacheco

DCC/FCUP

22/23

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))
```

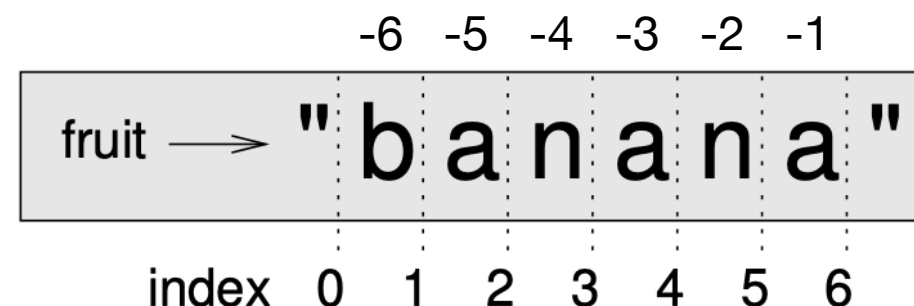
- ...ou como coleções de valores mais pequenos
- Tipos compostos são **objetos** que podem ter atributos, métodos e estado interno. Mais importante, **pode** haver efeitos secundários!



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("asdasd"):
    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'>
```