Programação I

Licenciatura em Engenharia Informática

2015–2016

Vitor Beires Nogueira

Escola de Ciências e Tecnologia
Universidade de Évora

Tuplos são imutáveis

Definição

Um tuplo é uma sequência de valores. Os valores podem ser de qualquer tipo e são indexados por inteiros.

Exemplo

```
>>> t = 'a', 'b', 'c', 'd', 'e'
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t1 = ('a')
>>> type(t1)
    <type 'str'>
>>> t2 = 'a',
>>> type(t2)
    <type 'tuple'>
```

Função tuple

Exemplo

```
>>> t = tuple()
>>> print(t)
    ( )
>>> t = tuple('hello')
>>> print(t)
    ('h', 'e', 'l', 'l', 'o')
>>> t[0]
    'h'
>>> t[1:3]
    ('e', 'l')
>>> t[0] = 'H'
    TypeError:  'tuple' object does not support item
    assignment
>>> t1 = ('H',) + t[1:]
```

Atribuição de valores a Tuplos

Exemplo (Troca de valores)

Considere que se pretende trocar os valores entre as variáveis `a` e `b`. Um hipótese seria:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Mais simples:

```
>>> a, b = b, a
```

O lado esquerdo é um tuplo de variáveis e o lado direito um tuplo de expressões

Exemplo (Atribuição mais elaborada)

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
>>> print(uname)
    monty
>>> print(domain)
    python.org
```

Tuplos como valor de retorno

- Uma função só pode retornar "uma coisa"
- Um tuplo é "uma coisa"!

Exemplo

Suponhamos que pretendemos dividir dois inteiros e calcular o quociente e o resto da divisão. A função `divmod` recebe dois argumentos e retorna um tuplo com dois valores (quociente e resto).

```
>>> t = divmod(7, 3)
>>> print(t)
(2, 1)
```

Será que podemos atribuir "directamente"o resto e o quociente?

```
>>> quot, rem = divmod(7, 3)
>>> print(quot)
2
>>> print(rem)
1
```

Função para calcular o mínimo e o máximo

```
def min_max(t):
    return min(t), max(t)
```

Tuplos como argumento de tamanho variável

- As funções podem ter um número variável de argumentos.
- Um parâmetro cujo nome começa por * *junta* (gather) todos os argumentos num tuplo.

Exemplo (printall)

```
def printall(*args):
    print(args)
```

- O operador * se utilizado como argumento, faz o complemento, ou seja, *dispersa* (scatter).

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
>>> divmod(*t)
(2, 1)
```

Defina uma função `sumall` que recebe um número variável de argumentos e devolve a sua soma.

Exemplo (sumall)

```
def sumall(* args):
    s = args[0]
    for i in range(1, len(args)):
        s += args[i]
    return s
```

Definição

A função `zip` é uma função do sistema que recebe duas ou mais sequências e "zippa-as" numa lista de tuplos, em que cada tuplo contém um elemento de cada sequência.

Exemplo (zip)

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

Exemplo (Atribuição de um tuplo num for)

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

Considere que pretendemos definir a função `has_match` que recebe duas sequências `t1` e `t2` e devolve `True` se existir um índice `i` tal que `t1[i] == t2[i]`

Exemplo (has_match)

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

Definição

O método `itens` dos dicionários devolve uma lista de tuplos, em que cada tuplo é um par chave-valor

Exemplo (itens)

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> print(t)
[('a', 0), ('c', 2), ('b', 1)]
```

Também podemos utilizar uma lista de tuplos para inicializar um novo dicionário:

```
>>> d = dict(zip('abc', range(3)))
>>> print(d)
{'a': 0, 'c': 2, 'b': 1}
```

Tuplos como chaves de dicionários

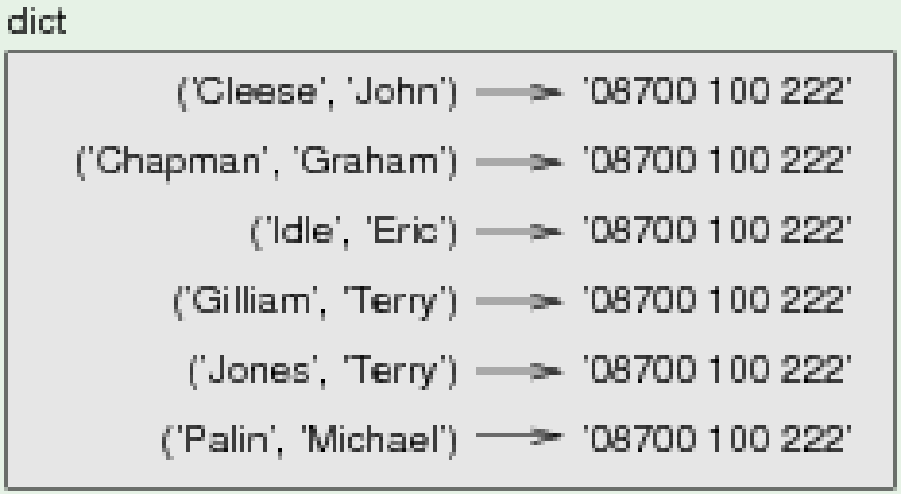
Os tuplos podem ser utilizados como chaves de dicionários.

Exemplo (Lista telefónica)

Uma lista telefónica pode ser interpretada como uma atribuição de uma par (Nome, Apelido) a um número de telefone

```
dic = {
    ('Cleese',  'John'): '08700 100 222',
    ('Chapman', 'Graham'): '08700 100 222'}

for last, first in dic:
    print(last, first, dic[last, first])
```



Comparando Tuplos

Os operadores relacionais (`==`, `<`, `...`) também funcionam com tuplos e outras sequências:

- o Python começa por comparar o primeiro elemento de cada sequência
- Enquanto os elementos forem iguais, avança para o seguinte até encontrar um diferente
- Elementos subsequentes não são considerados

Exemplo

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

O método `sort` das listas funciona de modo semelhante.

Ordenando palavras

Considere que pretendemos ordenar uma lista de palavras.

Exemplo (sort_by_length)

```
def sort_by_length(words):
    t = []
    for word in words:
        t.append((len(word), word))

    t.sort(reverse=True)

    res = []
    for length, word in t:
        res.append(word)
    return res
```

Sequências de sequências

- As strings, listas e tuplos são denominados por sequências (mas existem outras sequências)
- Já vimos que podemos ter listas de tuplos, ou seja, podemos ter sequências de sequências (existem outras combinações).
- Qual a sequência a utilizar?
- As strings tem algumas limitações: os elementos tem de ser caracteres e são imutáveis.
- As listas são mais usuais que os tuplos, uma vez que são mutáveis. No entanto por vezes podemos preferir tuplos:
  - Se quisermos utilizar uma sequência como chave de um dicionário só podemos utilizar um tipo imutável (como tuplo ou string)
  - Se pretendemos utilizar uma sequência como argumento de uma função podemos utilizar tuplos para reduzir o risco de comportamentos inesperados ("aliasing")