Curso de Computação 1
Introdução à Programação em Python

AULA 5

Listas e Dicionários

Na aula anterior estudamos os tipos de dados string e tupla. Nesta aula veremos mais dois tipos de dados bastante poderosos: as listas e os dicionários.

Nova ferramenta: A partir desta aula vamos trabalhar com mais uma ferramenta: **Python Tutor.** É uma ferramenta online e gratuita. Python Tutor ajuda a visualizar uma abstração do que está acontecendo na memória ao longo da execução de um código em Python. O que podemos visualizar com Python Tutor é muito similar ao teste de mesa, com a diferença de que ela faz a simulação da execução para nós. A vantagem, é que quando estamos com algum conceito errado na cabeça, ela ajuda, mostrando que o comportamento não é o que esperávamos, e é mais rápida. Mas ela não substitui o teste de mesa, no sentido que, se não somos capazes de simular a execução nós mesmos, teremos muita dificuldade de corrigir os erros que aparecerão no código. Ela será especialmente útil a partir de agora que nossos códigos estão ficando mais complexos e com a introdução dos tipos de dados mutáveis.
Veja como usar a ferramenta no vídeo:

https://youtu.be/7-MSDcyDjLk

Depois de assistir, experimente a ferramenta com os exemplos desse roteiro. Acesse o Python Tutor em:

http://www.pythontutor.com/

1. Estruturas de dados

Na ciência da computação, uma estrutura de dados é um modo particular de armazenamento e organização de dados em um computador de modo que possam ser usados eficientemente, facilitando sua busca e modificação.

A organização e os métodos para manipular essa estrutura lhe conferem singularidade e vantagens estratégicas, como a minimização do espaço ocupado na memória, além de (potencialmente) tornar o código mais enxuto e simples.

O termo *sequência* na computação denota uma estrutura de dados abstrata que implementa uma coleção de valores onde a ordem dos elementos importa, em que o mesmo valor pode ocorrer mais de uma vez. Cada ocorrência de um valor na sequência é normalmente chamada de item ou elemento. Se o mesmo valor ocorrer várias vezes, cada ocorrência é considerada um item distinto.
(texto adaptado de https://pt.wikipedia.org/wiki/Lista)

Em python, uma das implementações de *sequência* são as ***tuplas***. Nesta aula veremos outra: as ***listas***. As ***strings***, por sua vez, são um caso restrito de *sequência*, já que todos os seus elementos são necessariamente strings de tamanho 1.

2. Listas

Uma lista (list) em Python é uma sequência ou coleção ordenada de valores. Cada valor na lista é identificado por um índice. Os valores que formam uma lista são chamados elementos ou itens. Listas são similares a tuplas, com uma diferença: listas são mutáveis! Isso significa que podemos alterar pedaços da lista ou editá-la sem ter que criar outra lista. Esta característica faz com que este tipo de dado seja o mais versátil do Python.
Para criar uma lista, apresentamos seus elementos entre colchetes (**[ ]**), os separando por vírgula. Exemplo: **listanum=[1,2,3,4]**. Uma lista vazia é representada por **[ ].**

Da mesma forma que ocorre com strings e tuplas, a função **len** retorna o tamanho de uma lista (o número de elementos na lista). Para indexar uma lista, usamos o mesmo operador de indexação usado com tuplas e strings. Lembre-se que o índice do primeiro elemento é 0, índices negativos indicarão elementos da direita para a esquerda ao invés de da esquerda para a direita ("Jobim"[-1] é uma expressão que será avaliada como “m”).

Novamente, como em strings e tuplas, o operador **+** realiza a concatenação de listas, gerando uma nova lista. Analogamente, o operador ***** realiza a operação de replicação em uma lista um dado número de vezes. Por último, o operador **in** funciona também da mesma maneira, dizendo se um determinado valor é ou não um elemento da lista.

Exemplos:

```
>>> listaA = [’Ana’, ’Antonio’]
>>> listaB = [’Barbara’, ’Bernardo’, ’Bruno’]
>>> listaAB = listaA + listaB
>>>len(listaAB)
5
>>> listaAB
```

```
['Ana', 'Antonio', 'Barbara', 'Bernardo', 'Bruno']
>>> listaAB[2]     #Operacao de indexacao
'Barbara'
>>> 'Ana' in listaAB
True
>>> 'Joana' in listaAB
False
```

No vídeo a seguir você vai ver uma apresentação do tipo lista.
Introdução a listas

3. Mutabilidade e imutabilidade

Vamos entender agora as similaridades e diferenças entre eles e o papel de strings, tuplas e listas na construção de código em Python.

Similaridades:

Todos são *sequências* e podem ser indexados. A forma de indexar é a mesma: começando do zero até o tamanho-1. Índices negativos podem ser usados para o acesso na ordem reversa. Todos podem ser acessados através da operação de indexação (índice entre colchetes) e têm a mesma definição de "tamanho", que pode ser retornado pela função *len*.

Assim como dados de tipos simples (inteiros, floats, booleanos), todos podem servir para criação de novos dados, através de operações disponíveis para aquele tipo. Exemplos de operações destes tipos dados são concatenação (+), replicação (*) e fatiamento.

Diferenças:

Particularmente para as strings: todos os elementos de uma string são também strings de tamanho 1, ou seja, um caractere.
Exemplo:

```
>>> fruta = “banana”
>>> fruta[0]
”b”
```

Strings são um tipo de dados adequado para representar textos, frases, palavras e caracteres. Estes são conteúdos usuais de uma linguagem. Logo, elementos de comunicação entendidos por usuários e programadores.

Nas tuplas e listas, por sua vez, os elementos podem ser de qualquer tipo. Uma mesma tupla ou lista pode, inclusive, ter elementos de diversos tipos. Qual a diferença entre tuplas e listas, então?
A diferença entre tuplas e listas é a mutabilidade. Tuplas, como as strings, são tipos de dados imutáveis. Ou seja, qualquer alteração que seja necessária em um valor após a sua criação implica na criação de um novo dado, em algum outro lugar da memória. A criação deste novo dado não interfere na existência do dado original.

Por exemplo, imagine que foi criada uma tupla contendo os números 0, 2 e 3.

```
>>> tupla1 = (0,2,3)
```

Se agora precisamos trocar 0 do início da tupla pelo número 1, precisamos criar uma nova tupla com esse novo valor. Para fazer isso, precisamos fatiar a tupla existente para selecionar a parte que é preservada e em seguida concatená-la com o primeiro elemento desejado.

```
>>> tupla2 = (1,) + tupla1[1:]
>>> tupla2
(1,2,3)
>>> tupla1
(0,2,3)
```

A mesma coisa acontece com as strings. Para tirar um caractere de uma string, precisamos selecionar os trechos que serão mantidos e os trechos que serão excluídos e criar uma nova string com esses elementos.

```
>>> palavra = ‘bannana’
>>> palavra_corrigida = palavra[:2] +palavra[3:]
>>> palavra_corrigida
‘banana’
>>> palavra
‘bannana’
```

No caso das listas, podemos executar exatamente o mesmo processo para gerar novos dados, ou seja, construir novas listas a partir da desconstrução e reconstrução de uma lista original. Porém há outra maneira de lidar com listas, pelo fato de serem mutáveis.

O exemplo apresentado acima com tuplas funciona da mesma maneira com dados do tipo lista:

```
>>> lista1 = [0,2,3]
>>> lista2 = [1] + lista1[1:]
>>> lista1
[0,2,3]
>>> lista2
[1,2,3]
```

Mas essa solução só se justifica caso necessitemos preservar o valor original de lista1. Muitas vezes, porém, o que queremos é que o dado evolua ao longo do algoritmo de solução do problema. Nesses casos, não precisamos preservar o valor original e a solução usada acima não seria a solução mais indicada. A propriedade de mutabilidade das listas deve então ser usada, ficando assim:

```
>>> lista1 = [0,2,3]
>>> lista1[0] = 1
>>> lista1
[1,2,3]
```

Nesse caso, o valor [0,2,3] deixou de existir, pois internamente, na memória do computador a célula onde estava armazenado o valor 0 foi alterada para armazenar o valor 1.

Vamos conferir esses diferentes comportamentos no Python Tutor neste vídeo.

A lista é um tipo de dado mutável. Isso significa que podem ser atribuídos novos valores a uma lista existente sem uma nova lista que seja criada. A mesma lista inicial continua sendo utilizada, agora, com novos valores associados. É um tipo de dados muito útil em situações onde o valor associado a uma variável deve evoluir ao longo da execução.

Já as tuplas devem ser usadas em situações em que não há necessidade de adicionar, remover ou alterar elementos de um grupo de itens. Exemplos bons seriam os meses do ano, os dias da semana, as estações do ano, pontos do plano cartesiano. Os itens destes grupos não mudam, porém podem ser usados para gerar subconjuntos de itens específicos conforme problema que estamos tentando resolver.

Exemplo:
>> semana=(“segunda”, “terça”, “quarta”, “quinta”, “sexta”, "sábado", “domingo”)
>> fim_de_semana=semana[-2:]
>> fim_de_semana
>> (“sábado”, “domingo”)

4. Fatiamento de dados iteráveis

As listas, assim como as strings e as tuplas, podem ser fatiadas gerando sublistas. A operação de fatiamento é igual para strings, tuplas e listas: utilizando-se os colchetes após o nome da lista. Dentro do colchete definimos a posição inicial, a posição final (não inclusiva) e alternativamente um fator de incremento e decremento (passo), separados por dois pontos. Ou seja: *lista[inicio:fim:passo]*.

Vejamos um exemplo:

```
>>> numeros = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> alguns_numeros = numeros[2:7:2]
>>> alguns_numeros
[30, 50, 70]
```

Neste exemplo, foi criada uma sublista da lista “numeros” chamada “alguns_numeros”. A sublista começa no item 2 (lembre-se que a indexação começa em 0), vai até o item 7 (não inclusive), com incremento de 2.

O vídeo a seguir apresenta de forma detalhada o fatiamento de listas. Você pode observar que o funcionamento é exatamente o mesmo do que tínhamos visto para strings e tuplas.
Fatiamento de listas

O Python permite o uso do comando de atribuição para fatias de listas. Ficou curioso? Você pode pesquisar mais sobre isso na internet ;-). Temos também um vídeo que aborda esse assunto, para ajudar a matar a curiosidade: Atribuição a fatias.

Atividade: Vamos agora treinar um pouco com exercícios de fixação. Responda às perguntas da atividade “Uso de listas”. Fique atento ao prazo de entrega dessa atividade!

Curso de Computação 1
Introdução à Programação em Python

4. Dicionários

Além das sequências, o Python possui uma estrutura de dados chamada **dicionário** para coleções **não ordenadas.** Coleções não ordenadas, como o nome diz, não pressupõem ordem entre seus elementos, logo o conceito de índices de acesso não se aplica a elas.

Sobre a notação, enquanto strings usam aspas, tuplas usam parênteses, e listas usam colchetes, os dicionários são delimitados por chaves **{ }**. Apenas as chaves, **{}**, representam um dicionário **vazio**.

A principal diferença entre dicionários e as sequências (strings, tuplas e listas) está na forma como os elementos são acessados, uma vez que esse acesso não é feito através de uma posição (índice).

Como poderemos então acessar os elementos do dicionário, já que eles não têm um índice derivado de sua ordem? A verdade é que para usar dicionários fornecemos também nossa própria maneira de indexar os elementos, que não pressupõe uma ordem numérica. Os dicionários são estruturas mais sofisticadas que as sequências, já que eles armazenam pares chave-valor, ou seja, **mapeamentos**.

Cada elemento de um dicionário será na verdade um par chave-valor. Conceitualmente, os dicionários consistem de vários mapeamentos formados por pares de chave-valor, onde cada **valor** é referenciado através de sua **chave**. Portanto, para buscar um elemento em um dicionário, basta utilizar a chave associada ao mesmo.

O vídeo a seguir apresenta mais detalhes e como manipular dicionários. Introdução aos Dicionários

Suponha que você está desenvolvendo um programa para uma grande rede de supermercados. Nesse supermercado, cada item tem um preço associado a ele. Como você faria para armazenar os preços de cada produto dado que só sabemos o nome do mesmo quando chegamos ao caixa? A melhor forma de armazenar essas informações é em um dicionário!

Exemplo:

```
>>> produtos = {'farinha': 3.00,
```

```
'feijão': 5.00,
'leite': 4.25,
'açúcar': 2.49}
```

Cada chave nos permite acessar um valor, no exemplo acima, 'farinha', 'feijão', 'leite' e 'açúcar' são as **chaves** do dicionário enquanto os preços *float* são os **valores**. Para acessar um elemento em um dicionário, basta usar a variável que contém o dicionário seguida da chave entre colchetes:

Exemplos:

```
>>> produtos['farinha']
3.0
>>>produtos['feijão']
5.0
```

Para adicionar um novo elemento, como arroz, por exemplo, basta criar uma nova chave e atribuir um valor à mesma:

Exemplo:

```
>>> produtos['arroz'] = 6.90
```

Acessando o valor contido em 'produtos', podemos ver que agora o par, aqui chamado de **item**, *(arroz, 6.9)* pertence ao dicionário.

Exemplo:

```
>>> produtos
{'farinha': 3.0, 'feijão': 5.0, 'leite': 4.25,  'açúcar': 2.49,
'arroz': 6.9}
```

Diferente das listas, não precisamos que uma posição exista previamente para adicionar um novo elemento a um dicionário. Uma coisa interessante é que não há garantia de que as chaves estarão ordenadas, pois o dicionário é um conjunto não ordenado. Como os valores são acessados através de suas respectivas chaves, a ordem dos elementos não é importante.

Se tentarmos acessar um produto que não está cadastrado no nosso supermercado, ou seja, não pertence ao dicionário, é retornado um erro.

Exemplo:

```
>>>produtos['sal']
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
```

```
    produtos['sal']
KeyError: 'sal'
```

O operador **IN** também pode ser utilizado para verificar se uma chave pertence a um dicionário.

Exemplos:

```
>>>'sal' in produtos
False
>>>'farinha' in produtos
True
```

As **chaves dos dicionários** são dados de tipo imutável, geralmente strings. Também podem ser tuplas ou tipos numéricos. Já os valores em um dicionário podem ser de qualquer tipo. Veja o que acontece se tentarmos usar uma lista como chave.

Exemplo:

```
>>>dicionario = {['farinha', 'açúcar', 'leite']: 'bolo'}
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    dicionario = {['farinha', 'açúcar', 'leite']: 'bolo'}
TypeError: unhashable type: 'list'
```

Não podemos usar listas como chaves, mas estas podem ser utilizadas como valores do dicionário.

Exemplo:

```
>>> dicionario = {'bolo': ['farinha', 'açúcar', 'leite']}
>>> dicionario
{'bolo': ['farinha', 'açúcar', 'leite']}
```

Já as tuplas podem ser utilizadas como chaves.

Exemplo:

Podemos trabalhar com um dicionário de localização de cidades ou pontos de interesse. As localizações serão representadas por uma tupla de 2 elementos, latitude e longitude, recuperadas a partir de um gps. Cada um desses elementos, por sua vez, é também uma tupla (com três elementos).

Considerando a latitude e a longitude da localização da cidade do Rio de Janeiro (informação disponível em http://www.rio.rj.gov.br/web/riotur/caracteristicas-geograficas):

```
>>> o_que_tem_em  = {((25,54,33),(43,10,21)): 'Rio de Janeiro'}
>>> o_que_tem_em [((25,54,33),(43,10,21))]
'Rio de Janeiro'
```

Dicionários são **mutáveis** e cada **chave é única**, não é possível ter duas chaves iguais apontando para valores diferentes. Para modificar o valor de uma chave, basta utilizar a mesma sintaxe da inserção. Suponha que o preço do arroz aumentou. Para alterá-lo, basta utilizar a chave 'arroz' e atribuir o novo valor.

Exemplo:

```
>>>produtos['arroz']
6.9
>>>produtos['arroz'] = 7.50
>>>produtos['arroz']
7.5
>>>produtos
{'farinha': 3.0, 'feijão': 5.0, 'leite': 4.25, 'açúcar': 2.49,
'arroz': 7.5}
```

Atividade: Responda às perguntas do formulário "Uso de Dicionários''. Lembre que você pode testar o código no IDLE ou no Pythontutor.

5. Prática em Programação

Após concluir as etapas anteriores deste roteiro, faça as atividades práticas desta aula, disponíveis no Google Classroom da turma.

Até a próxima aula!