3

TEXTO BASE

PROGRAMAÇÃO ORIENTADA A OBJETOS

Faculdade IMPACTA

Texto base

3

Listas e Dicionários em Python

Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

Neste capítulo vamos fazer uma breve revisão de sequências (listas e tuplas) e das estruturas básicas de controle de fluxo: decisão e repetição. Além disso, vamos aprender sobre dicionários em Python, estruturas que facilitam a manipulação de dados dentro de um programa e são extremamente úteis em diversas situações. Praticamente todas as linguagens modernas possuem estruturas semelhantes ou análogas.

3.1. Sequências

O Python possui três tipos básicos de sequências: listas, tuplas e intervalos (*range*), e mais dois tipos feitos especificamente para lidar com sequências de caracteres (*strings*) e de dados binários. Nesta aula vamos revisar as sequências de lista e tupla.

A principal diferença entre elas é que a lista é uma sequência mutável, cujos itens podem ser alterados, e a tupla é imutável, ou seja, após criada não pode mais ser alterada. Lembre-se que aqui estamos falando dos objetos na memória, e não do conteúdo de uma variável, pois sempre podemos atribuir um novo valor a uma variável, sobrescrevendo o valor anterior.

A lista completa de operações que podemos fazer em todas as sequências é chamada de Operações Comuns de Sequências (PSF[1], 2021a). Para as listas, temos ainda as operações listadas em Tipos de Sequências Mutáveis (PSF, 2021b). Não deixe de conferir na bibliografia o link para a documentação do Python em cada caso.

Faremos aqui uma breve revisão de listas e tuplas em Python, recuperando os conceitos vistos em Linguagem de Programação.

[1] PSF - *Python Software Foundation*

3.1.1. Listas

Uma lista em Python é uma estrutura de dados linear, ordenada, mutável e heterogênea, isto é, os itens de uma lista tem uma posição fixa, podem ser modificados e podem ser de tipos diferentes entre si. Ela é representada em Python por colchetes, sendo os itens separados por vírgula. Vejamos alguns exemplos:

```
>>> inteiros = [1, 10, 3]
>>> compras = ['cereal', 'suco', 'banana', 'maçã', 'azeite']
>>> lista_mista = [5, 'bla', True, 10.3]
>>> lista_vazia = []
```

Cada elemento adicionado em uma lista ganha automaticamente um índice, referente à sua posição na lista (primeiro, segundo, etc.). Em Python, os índices começam no zero e aumentam constantemente para a direita conforme necessário. Há também um índice negativo, que começa em -1, a partir do final da lista, e diminui conforme caminhamos de volta para o começo da lista, como mostra a Figura 3.1.

Figura 3.1: Visualização dos índices de uma lista em Python

	['cereal',	'suco',	'banana',	'maçã',	'azeite']
índices positivos	0	1	2	3	4
índices negativos	-5	-4	-3	-2	-1

Fonte: do autor, 2021

Veremos a seguir algumas das ações que podemos realizar em uma lista, confira a lista completa na documentação do Python (PSF, 2021a; PSF, 2021b).

3.1.1.1. Acessar um item da lista

Para acessar um elemento da lista, usamos também um par de colchetes com o índice do elemento em questão.

```
>>> inteiros[1]
10
>>> compras[0]
'cereal'
```

Lembrando que os índices começam em zero, então se tentarmos acessar o quinto elemento da lista de compras definida acima, devemos usar o índice 4.

```
>>> compras[4]
'azeite'
```

Se tentarmos usar o valor 5 para o índice, iremos receber um erro dizendo que o valor está fora do intervalo de índices da lista.

```
>>> compras[5]
(...)
IndexError: list index out of range
```

Já o índice negativo nos permite acessar de maneira fácil o último elemento de uma lista, sem precisarmos nos preocupar com o tamanho da lista.

```
>>> compras[-1]
'azeite'
```

3.1.1.2. Substituir um item da lista

As posições na lista funcionam de maneira análoga a uma variável, então podemos simplesmente atribuir um valor a um elemento existente, sobrescrevendo assim o valor anterior:

```
>>> compras[1] = 'limão'
>>> compras
['cereal', 'limão', 'banana', 'maçã', 'azeite']
```

3.1.1.3. Inserir um item em uma dada posição na lista

Para inserir um item na lista usamos o método `insert`, passando a ele como argumentos o índice e o valor a ser inserido na lista. Em breve ficará mais claro como funcionam os métodos de um objeto, por hora, precisamos saber que devemos chamá-los a partir do objeto que queremos alterar, usando a notação de ponto:

```
>>> compras.insert(1, 'pão')
>>> compras
['cereal', 'pão', 'limão', 'banana', 'maçã', 'azeite']
```

Observe que foi inserido um novo valor na segunda posição (índice 1), e que todos os valores da lista a partir dessa posição foram deslocados para a direita.

3.1.1.4. Remover um item da lista

Existem três formas de se remover um item de uma lista:

1) Com o comando `del`: esta palavra chave deleta um objeto da memória, podemos usá-la também para deletar uma variável por exemplo, então basta acessar o item que queremos excluir e passá-lo ao comando del:

```
>>> del compras[-1]
>>> compras
['cereal', 'pão', 'limão', 'banana', 'maçã']
```

2) Com o método `pop`: este método recebe um índice como argumento, exclui o respectivo item e **retorna** o seu valor, caso nenhum índice seja passado, por padrão remove e retorna o último item:

```
>>> compras.pop(3)
'banana'
>>> compras
['cereal', 'pão', 'limão', 'maçã']
```

3) Com o método `remove`: este método recebe um valor e faz uma busca na lista, removendo o primeiro item que corresponder ao valor passado. Se houverem itens repetidos, apenas o primeiro é removido:

```
>>> compras.remove('pão')
>>> compras
['cereal', 'limão', 'maçã']
```

3.1.1.5. Acrescentar um item ao final da lista

Para acrescentar um item ao final de uma lista, usamos o método `append`, que recebe um objeto e o adiciona ao final da lista:

```
>>> compras.append('sorvete')
>>> compras
['cereal', 'limão', 'maçã', 'sorvete']
```

VAMOS PRATICAR!

1) Tente usar o `append` para adicionar uma lista ao final de outra lista e veja o que acontece.

2) Agora refaça o teste usando o método `extend` e compare os resultados.

3.1.1.6. Concatenar duas listas

Podemos também concatenar duas listas usando o operador `+`, de maneira análoga a concatenação de strings:

```
>>> compras + inteiros
['cereal', 'limão', 'maçã', 'sorvete', 1, 10, 3]
```

Na concatenação, uma nova lista é gerada e retornada pela operação, sem que as listas originais sofram qualquer alteração[2].

[2] Há uma exceção à essa regra quando usamos o operador de atribuição composta `+=`, com ele, o Python irá alterar o mesmo objeto na memória, de maneira análoga à utilização do método `extend`.

3.1.2. Tuplas

Assim como listas, tuplas são estruturas de dados lineares, ordenadas e heterogêneas, no entanto, ao contrário de listas, tuplas são **imutáveis**. Em Python, as tuplas são definidas por um par de parênteses, com os itens separados por vírgula.

```
>>> tupla = (1, 10, 3)
>>> tupla_vazia = ()
```

Para definir uma tupla com um único elemento, devemos colocar uma vírgula extra para que os parênteses não sejam interpretados com uma expressão aritmética.

```
>>> tupla_de_um_item = (8,)
>>> nao_eh_uma_tupla = (5)
```

Ao separar elementos por vírgula, mesmo sem os parênteses, o Python também irá criar uma tupla.

```
>>> nova_tupla = 14, 25, 91
>>> nova_tupla
(14, 25, 91)
```

Como tuplas não podem ser modificadas, apenas os métodos comuns a todas as sequências são aplicáveis. Dos exemplos visto em lista, podemos apenas acessar um valor da tupla e concatenar duas tuplas, pois em nenhum dos casos os objetos originais são modificados. Da mesma forma que em listas, acessar um item inexistente irá gerar um erro de execução.

```
>>> tupla[1]
10
>>> tupla + tupla_de_um_item
(1, 10, 3, 8)
```

Dica: essas mesmas duas operações também funcionam com *strings*, que são um tipo especial de sequência imutável. Para aprender mais sobre as estruturas de dados em Python veja o tutorial oficial (PSF, 2021c).

3.1.3. Operações de pertencimento em listas, tuplas e *strings*

O operador de pertencimento tem a mesma precedência dos operadores relacionais e pode ser usado para todos os tipos de sequências, verificando se um objeto está ou não contido na sequência.

```
>>> tupla = (1, 10, 3)
>>> compras = ['cereal', 'limão', 'maçã', 'sorvete']
>>> texto = 'Olá mundo!'
>>> 3 in tupla
True
>>> 'uva' not in compras
True
>>> 'M' in texto
False
```

Observe que a negação do operador retorna `True` quando o objeto não é encontrado na lista, e que a letra M maiúscula de fato não existe na *string* da variável `texto`, pois o Python diferencia letras maiúsculas de minúsculas tanto em identificadores (nomes de variáveis, funções, etc.) quanto em *strings*.

3.1.4. Desempacotamento de sequências

O desempacotamento de sequências (listas, tuplas e conjuntos[3]), é uma forma de atribuir os itens de uma sequência a diferentes variáveis, que pode ser usado com o operador de atribuição, em laços do tipo `for` e na passagem de argumentos para funções. Vejamos o uso com o operador de atribuição:

```
>>> a, b = [1, 2]
>>> a
1
>>> b
2
```

No exemplo anterior, o número de itens precisa ser igual ao número de variáveis e vice-versa, caso contrário o Python irá levantar um erro dizendo que não foi possível realizar o desempacotamento.

Se tivermos uma sequência cujos itens sejam em si uma sequência, podemos usar esse recurso também em laços do tipo `for`. Imagine a seguinte lista, onde cada item é uma tupla com um par `(<letra>, <número>)`:

```
>>> lista = [('a', 1), ('b', 2), ('c', 3)]
```

Iterando sobre esta lista com um `for` tradicional, podemos fazer:

[3] Uma sequência que não possui itens repetidos, representada por chaves em Python. Veja mais em https://docs.python.org/3/tutorial/datastructures.html#sets.

```
>>> for item in lista:
...     print(f'{item} --> {item[0]}: {item[1]}')

('a', 1) --> a: 1
('b', 2) --> b: 2
('c', 3) --> c: 3
```

Ou seja, se quisermos acessar a letra e o número em cada item, precisamos usar os índices como no exemplo anterior, mas com o desempacotamento, podemos fazer:

```
>>> for letra, numero in lista:
...     print(f'{letra}: {numero}')

a: 1
b: 2
c: 3
```

Agora podemos acessar diretamente a letra e o número de cada item dentro do `for`, que estão disponíveis em variáveis que deixam o código mais legível, quando comparamos a `item[0]` e `item[1]`. Observe que esse desempacotamento ocorre no momento de atribuição de valor que é realizado pelo laço `for`.

Para realizar o desempacotamento para os argumentos de uma função, devemos usar um * antes da sequência que será desempacotada na chamada da função:

```
>>> def teste_desempacotamento(a, b, c):
...     print(f'{a=}, {b=}, {c=}')

>>> sequencia = 35, 21, 9
>>> teste_desempacotamento(*sequencia)
a=35, b=21, c=9
```

3.2. Estruturas de controle de fluxo

São as estruturas que nos permitem alterar o fluxo sequencial de execução do código, seja escolhendo um determinado caminho em função da avaliação de uma condição (estruturas de seleção), seja repetindo um trecho de código (estruturas de repetição).

3.2.1. Estruturas de seleção

Em Python, a estrutura de seleção é feita com o comando `if`, seguido de uma condição. Caso a condição seja verdadeira, o bloco de código definido por esse comando será executado, e caso seja falsa, será ignorado.

```
if <condição>:
    <bloco de código>
```

O guia de estilo recomenda não utilizar parênteses em torno da condição e caso seja utilizada uma flag booleana, não é recomendado a comparação com os tipos `True` e `False`, deve-se usar diretamente a flag com o operador `not` se necessário: `if flag:` ou `if not flag:`, respectivamente.

É possível também definir um bloco de código para ser executado quando a condição do comando `if` é avaliada para `False`, com o comando `else`.

```
if <condição>:
    <bloco de código se verdadeira>
else:
    <bloco de código se falsa>
```

Observe que não há nenhuma condição após o comando `else`, pois este bloco irá se e somente se a condição do comando `if` for falsa.

Há ainda o comando `elif`, que é uma contração dos comandos `else` + `if`, quando temos laços encadeados, com o objetivo de reduzir uma indentação excessiva.

```
if <condição C1>:
    <bloco de código se verdadeira>
else:
    if <condição C2>:
        <bloco de código se C1 verdadeira e C2 falsa>
    else:
        <bloco de código se C1 e C2 falsas>
```

Podemos reescrever o código do exemplo anterior usando o comando `elif`, da seguinte forma:

```
if <condição C1>:
    <bloco de código se verdadeira>
elif <condição C2>:
    <bloco de código se C1 verdadeira e C2 falsa>
else:
    <bloco de código se C1 e C2 falsas>
```

Observe que os blocos de execução são exatamente os mesmos, mas agora temos apenas um nível de indentação, independentemente de quantos comandos `elif` houver. Veja o seguinte exemplo, retirado do tutorial oficial do Python (PSF, 2021d).

```
>>> x = int(input("Por favor entre um inteiro: "))
Por favor entre um inteiro: 42
>>> if x < 0:
...     x = 0
...     print('Negativo alterado para zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Unitário')
... else:
...     print('Mais que um')
```

3.2.2. Estruturas de repetição

As estruturas de repetição podem ser divididas em dois grupos, indefinidas e definidas. No primeiro, não sabemos a priori quantas vezes a instrução será executada pois dependemos da avaliação de uma condição. Já no segundo, estamos iterando sobre uma sequência, portanto o número de repetições será definido pelo seu tamanho, e as instruções executadas uma vez para cada valor da sequência.

3.2.2.1. Estruturas de repetição indefinidas

As estruturas de repetição indefinidas são feitas com o comando `while`:

```
while <condição>:
    <bloco de código>
```

Esta estrutura é muito parecida com a estrutura de seleção simples (o comando `if`), mas a principal diferença é que, ao contrário do `if`, após a execução do bloco de código, o `while` irá reavaliar a condição e o processo se repete enquanto esta avaliação resultar em `True`.

```
soma = 0
while soma < 21:
    carta = int(input('Digite o valor da carta: '))
    soma += carta

print(f'Seu resultado é {soma}')
```

No exemplo acima, é pedido o valor de uma carta e este valor é acumulado em uma variável soma, caso esta soma seja maior ou igual a 21, a condição resultará em falso e o laço é encerrado. Em outras palavras, enquanto a soma for menor que 21, será pedido o valor de mais uma carta e o processo se repete.

VAMOS PRATICAR!

1) Este exemplo do comando `while` simula parcialmente o comportamento do jogo 21 (black jack), altere-o para completar o comportamento esperado desse jogo incluindo a possibilidade de o jogador decidir parar de pegar cartas por opção, independentemente do valor que ele já tenha acumulado em soma. Lembre-se que o jogo deve continuar parando caso a soma ultrapasse 21.

2) Crie uma solução alternativa para a questão anterior. Caso você tenha usado o comando `break` para resolvê-lo, tente agora chegar à mesma solução sem utilizar tal comando. Caso não tenha utilizado o `break`, tente alterar seu código para utilizá-lo.

3.2.2.2. Estruturas de repetição definidas

As estruturas de repetição definidas são feitas com o comando `for`:

```
for <variável> in <sequência>:
    <bloco de código>
```

O bloco de código de um laço `for` será executado uma vez para cada valor da sequência, com o valor da vez sendo atribuído à variável no início do laço. Isto é, ao começar um laço `for`, o Python irá atribuir o primeiro valor da sequência à variável definida no laço, executar o bloco de código e repetir o processo enquanto houver valores na sequência. Após o último valor, o laço é encerrado automaticamente e segue-se o fluxo de execução.

Execute o código do exemplo a seguir para ver o funcionamento deste laço.

```
lista = ['a', 1, True, 3.5]
for valor in lista:
    print(f'valor: {valor}, do tipo: {type(valor)}')

print('-----\nfim').
```

Uma função muito utilizada em conjunto com os laços definidos é a função `range`, que cria um intervalo de números inteiros em Python. Esta função pode receber 1, 2 ou 3 parâmetros, que devem sempre ser números inteiros. Ao receber um único parâmetro, é gerada uma sequência começando em zero e indo até o número anterior ao número dado, ou seja, o número dado será o tamanho da sequência.

```
>>> for i in range(3):
...     print(i)
...
0
1
2
```

E com isso podemos usar tal sequência como os índices de uma lista ou sequência, em combinação com a função `len`, que nos dá o tamanho de uma sequência.

```
lista = ['a', 1, True, 3.5]
for i in range(len(lista)):
    print(i, lista[i])
```

Para saber mais, veja a explicação mais detalhada no tutorial oficial do Python, cujo link está na bibliografia (PSF, 2021e).

3.3. Dicionários

Dicionários são uma estrutura de mapeamento, atualmente a única estrutura padrão desse tipo em Python, mas o que é uma estrutura de mapeamento? Uma estrutura de mapeamento cria uma associação entre dois objetos, uma chave e um valor. Em outras linguagens, estruturas semelhantes são chamadas de *hashmaps*, *hash tables* ou arrays associativos.

Para criar um dicionário em Python, colocamos, entre chaves, uma lista de pares `chave: valor` separados por vírgula. Vejamos um exemplo:

```
>>> dicionario_vazio = {}
>>> notas = {'Jack': 8.3, 'Anna': 9.0, 'Cris': 7.5}
```

Essa é a forma literal de se criar um dicionário em Python, mas há diversas outras formas que podem ser mais vantajosas em algumas situações, em especial quando precisamos converter dados entre diferentes tipos de estruturas. Estas formas podem ser vistas na documentação (PSF, 2021f), como por exemplo criar um dicionário a partir de uma lista de tuplas com dois valores.

O exemplo a seguir cria o mesmo dicionário atribuído a variável `notas` no exemplo anterior, só que a partir de uma lista já existente, usando a função `dict()`.

```
>>> lista_notas = [('Jack', 8.3), ('Anna', 9.0), ('Cris', 7.5)]
>>> notas_2 = dict(lista_notas)
>>> notas == notas_2
True
```

As chaves em um dicionário precisam ser únicas, então ao criarmos dois itens com a mesma chave, o valor do segundo item sobrescreve o do primeiro.

```
>>> notas = {'Jack': 8.3, 'Anna': 9.0, 'Jack': 7.5}
{'Jack': 7.5, 'Anna': 9.0}
```

Podemos pensar na chave como se fosse o “índice” do valor a que estamos nos referindo, mas agora temos a liberdade de criar esses “índices”. Os dicionários só

aceitam como chave objetos que sejam imutáveis, como por exemplo *strings*, inteiros, *floats* e tuplas cujos itens sejam também imutáveis, mas vale ressaltar que não é uma boa ideia usar um *float* como chave pois, devido a sua natureza, só é possível guardar um valor aproximado dele na memória, o que pode levar a erros ou inconsistências. As chaves mais comumente usadas são *strings* e inteiros.

Quanto aos valores de um dicionário, não há nenhuma restrição de tipo, então podemos guardar quaisquer objetos do Python, inclusive outros dicionários. Ainda, os valores podem ser repetidos sem nenhum problema.

Veremos agora alguns métodos de dicionários, a lista completa pode ser encontrada na documentação (PSF, 2021f).

3.3.1. Acessar um valor do dicionário

Para acessar os valores de um dicionário, de maneira análoga a listas ou tuplas, devemos indicar o valor da chave entre colchetes:

```
>>> notas['Jack']
7.5
>>> notas['Anna']
9.0
```

Se tentamos acessar uma chave que não existe no dicionário, obtemos um erro:

```
>>> notas['Megan']
(...)
KeyError: 'Megan'
```

Outra forma de acessar o valor de um dicionário é usar o método get, cuja sintaxe é `dicionario.get(key, default=None)`, e ele retorna o valor associado à chave, ou o valor `default` caso a chave não exista.

```
>>> notas.get('Megan')
>>> notas.get('Megan', 'Nome não encontrado')
'Nome não encontrado'
```

Observe que a primeira chamada retorna `None`, portanto a *Shell* não exibe nada.

3.3.2. Inserir um valor no dicionário

Ao contrário de listas, para acrescentar um novo valor ao dicionário podemos fazer uma atribuição diretamente a uma chave, sendo ela existente ou não.

```
>>> notas['Megan'] = 8.0
>>> notas
{'Jack': 7.5, 'Anna': 9.0, 'Megan': 8.0}
```

Novos itens são sempre inseridos no final, pois desde a versão 3.7 do Python, os dicionários guardam a ordem de inserção dos pares chave-valor. Caso a chave já exista, o seu valor será sobrescrito e a ordem não é alterada.

```
>>> notas['Jack'] = 6.0
>>> notas
{'Jack': 6.0, 'Anna': 9.0, 'Megan': 8.0}
```

Para atualizar diversos valores de uma única vez, podemos usar o método `update`, que pode receber como argumento uma lista de tuplas, como vimos na criação de um dicionário com a função `dict`, ou então um outro dicionário.

Caso haja chaves repetidas, os valores mais à direita terão prioridade, pois serão inseridos por último e irão sobrescrever os anteriores.

```
>>> inteiros = {}
>>> inteiros.update({1: 'um', 2: 'dois'})
>>> inteiros
{1: 'um', 2: 'dois'}
>>> inteiros.update([(3, 'três'), (4, 'quatro')])
>>> inteiros
{1: 'um', 2: 'dois', 3: 'três', 4: 'quatro'}
```

Na versão 3.9 do Python, foi introduzido o operador de união para dicionários também (este operador já existia para conjuntos em Python - *set*), representado pelo caractere de barra vertical (*pipe*): `|`.

```
>>> d1 = {'a': 1, 'b': 2}
>>> d2 = {'c': 3: 'd': 4}
>>> d1 | d2
{'a': 1, 'b': 2, 'c': 3: 'd': 4}
```

3.3.3. Excluir um item do dicionário

Para excluir um par chave-valor do dicionário podemos acessar o objeto e passá-lo ao comando del, como vimos em listas, e caso a chave não exista, objetos um `KeyError`.

```
>>> del inteiros[2]
>>> inteiros
{1: 'um', 3: 'três', 4: 'quatro'}
```

Ou podemos usar o método `pop`, que nos retorna o valor associado à chave e exclui o par chave-valor do dicionário.

```
>>> inteiros.pop(4)
'quatro'
>>> inteiros
{1: 'um', 3: 'três'}
```

Caso a chave não esteja no dicionário, também obtemos um `KeyError`. Para evitar o erro, podemos passar mais um argumento para o método, que será usado como valor padrão a ser retornado quando a chave não existir.

```
>>> inteiros.pop(5)
(...)
KeyError: 5
>>> inteiros.pop(5, 'chave não encontrada')
'chave não encontrada'
```

3.3.4. Operações de pertencimento em um dicionário

Os operadores de pertencimento, quando usados em um dicionário, fazem a verificação na lista de chaves do dicionário. Podemos então usá-los para verificar se uma chave existe ou não em um dado dicionário.

```
>>> 5 not in inteiros
True
>>> 'Jonas' in notas
False
```

Como os valores associados às chaves podem ser qualquer objeto do Python, não há um método padrão para fazer uma verificação da existência de valores no dicionário. Quando isso for necessário, devemos escrever nossos próprios métodos ou funções para tal. Ao aprendermos a criar classes poderemos usar o conceito de herança para estender um tipo do Python, criando nosso próprio tipo personalizado com recursos e funcionalidades extras.

3.3.5. Métodos especiais para iterar sobre um dicionário

É comum precisarmos iterar sobre todos os itens de um dicionário, e usando um laço for, podemos fazer isso iterando chave a chave. Crie um arquivo “dicionarios.py”, na pasta “aula03”, com o seguinte código:

```
notas = {'Jack': 6.0, 'Anna': 9.0, 'Megan': 8.0}
for chave in notas:
    print(chave, notas[chave])
```

Agora, com o VSCode aberto na pasta da disciplina[4], execute no terminal:

```
> python aula03/dicionarios.py
Jack 6.0
Anna 9.0
Megan 8.0
```

Neste exemplo, estamos iterando sobre as chaves do dicionário, e precisamos acessar o elemento usando a notação `dicionario[chave]`. Isso é uma operação bastante eficiente, devido a natureza dos dicionários em Python, mas há uma forma ainda melhor de fazer esta iteração, com os métodos `dict.keys()`, `dict.values()` e `dict.items()`. Estes métodos retornam um objeto especial do Python chamado Objeto de Visualização de Dicionários (PSF, 2021g), que nos fornece uma visão dinâmica sobre as entradas do dicionário.

[4] Passamos para o comando “python” o caminho relativo do arquivo que queremos executar. Se você abriu o VSCode na pasta da aula, altere o comando para conter apenas o nome do arquivo, ou se preferir, passe o caminho absoluto para o arquivo.

Podemos pensá-los como “listas” que podem ser iteradas e nas quais podemos fazer operações de pertencimento. Altere o laço for do arquivo “dicionarios.py” para:

```
for nome, nota in notas.items():
    print(nome, nota)
```

Ao executar o arquivo, o resultado obtido será o mesmo, mas o código está muito mais legível, uma vantagem ainda maior quando trabalhamos com situações mais complexas que a do exemplo.

O método `dict.keys()` retorna uma visualização das chaves do dicionário, o `dict.values()` dos valores, e o `dict.items()` dos pares chave-valor. Para ver o efeito destes métodos na *Shell* podemos converter o retorno para uma lista estática:

```
>>> notas = {'Jack': 6.0, 'Anna': 9.0, 'Megan': 8.0}
>>> list(notas.keys())
['Jack', 'Anna', 'Megan']
>>> list(notas.values())
[6.0, 9.0, 8.0]
>>> list(notas.items())
[('Jack', 6.0), ('Anna', 9.0), ('Megan', 8.0)]
```

Observe que essa conversão para lista não é necessária quando estivermos usando estes métodos em um laço `for`, e não deve ser feita, pois reduz a legibilidade do código e retira a dinamicidade do objeto original, podendo afetar seu desempenho.

Referências

PSF. **Built-in types:** common sequence operations. 2021a. Disponível em: <https://docs.python.org/3/library/stdtypes.html#common-sequence-operations>. Acesso em: 03 fev. 2021.

PSF. **Built-in Types:** mutable sequence types. 2021b. Disponível em: <https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types>. Acesso em: 03 fev. 2021.

PSF. **The python tutorial:** data structures. 2021c. Disponível em: <https://docs.python.org/3/tutorial/datastructures.html>. Acesso em: 03 fev. 2021.

PSF. **The python tutorial:** if statements. 2021d. Disponível em: <https://docs.python.org/pt-br/3/tutorial/controlflow.html#if-statements>. Acesso em: 25 jun. 2021.

PSF. **The python tutorial:** a função range(). 2021e. Disponível em: <https://docs.python.org/pt-br/3/tutorial/controlflow.html#the-range-function>. Acesso em: 25 jun. 2021.

PSF. **Built-in types:** mapping types - dict. 2021f. Disponível em: <https://docs.python.org/3/library/stdtypes.html#mapping-types-dict>. Acesso em: 03 fev. 2021.

PSF. **Built-in types:** dictionary view objects. 2021g. Disponível em: <https://docs.python.org/3/library/stdtypes.html#dict-views>. Acesso em: 04 fev. 2021.