```
if(acertou):
    print('Você ganhou!!')
else:
    print('Você perdeu!!')
print('Fim do jogo')
```



## 5.2 SEQUÊNCIAS

Desenvolvemos um novo jogo e conhecemos uma nova estrutura, as listas. Uma lista é denominada por uma sequência de valores, e o Python possui outros tipos de dados que também são sequências. Neste momento, conheceremos um pouco de cada uma delas.

Sequências são *containers*, um tipo de dado que contém outros dados. Existem três tipos básicos de sequência: `list` (lista) , `tuple` (tupla) e `range` (objeto de intervalo). Outro tipo de sequência famoso que já vimos são as *strings* que são sequências de texto.

Sequências podem ser mutáveis ou imutáveis. Sequências imutáveis não podem ter seus valores modificados. Tuplas, *strings* e *ranges* são sequências imutáveis, enquanto listas são sequências mutáveis.

As operações na tabela a seguir são suportadas pela maioria dos tipos de sequência, mutáveis e imutáveis. Na tabela abaixo, s e t são sequências do mesmo tipo, n , i , j e k são inteiros e x é um objeto arbitrário que atende a qualquer tipo e restrições de valor impostas por s .

| Operação | Resultado |
|---|---|
| x in s | True se um item de s é igual a *x* |
| x not in s | False se um item de s é igual a *x* |
| s + t | Concatenação de s e s |
| s *n ou n* s | Equivalente a adicionar s a si mesmo n vezes |
| s[i] | Elemento na posição i de s |

| | |
|---|---|
| s[i:j] | Fatia s de i para j |
| s[i:j:k] | Fatia s de i para j com o passo k |
| len(s) | Comprimento de s |
| min(s) | Menor item de s |
| max(s) | Maior item de s |
| s.count(x) | Número total de ocorrências de x em s |

- **Listas**

Uma lista é uma sequência de valores onde cada valor é identificado por um índice iniciado por 0. São similares a *strings* (coleção de caracteres) exceto pelo fato de que os elementos de uma lista podem ser de qualquer tipo. A sintaxe é simples, listas são delimitadas por colchetes e seus elementos separados por vírgula:

```
>>> lista1 = [1, 2, 3, 4]
>>> lista1
[1, 2, 3, 4]
>>>
>>>lista2 = ['python', 'java', 'c#']
>>> lista2
['python', 'java', 'c#']
```

O primeiro exemplo é uma lista com 4 inteiros, o segundo é uma lista contendo três *strings*. Contudo, listas não precisam conter elementos de mesmo tipo. Podemos ter listas heterogêneas:

```
>>> lista = [1, 2, 'python', 3.5, 'java']
>>> lista
[1, 2, 'python', 3.5, 'java']
```

Nossa *lista* possui elementos do tipo `int`, `float` e `str`. Se queremos selecionar um elemento específico, utilizamos o operador `[ ]` passando a posição:

```
>>> lista = [1, 2, 3, 4]
>>> lista[0]
1
>>> lista[1]
2
>>> lista[2]
3
>>> lista[3]
4
>>> lista[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Quando tentamos acessar a posição 4 fazendo `lista[4]`, aparece o erro `IndexError`, dizendo que a lista excedeu seu limite já que não há um quinto elemento (índice 4).

O Python permite passar valores negativos como índice que vai devolver o valor naquela posição de

forma reversa:

```
>>> lista = [1, 2, 3, 4]
>>> lista[-1]
4
>>> lista[-2]
3
```

Também é possível usar a função `list()` para criar uma lista passando um tipo que pode ser iterável como uma *string*:

```
>>> lista = list('python')
>>> lista
['p', 'y', 't', 'h', 'o', 'n']
```

Listas são muito úteis, por exemplo:

```
    meses = ['Janeiro', 'Fevereiro', 'Março', 'Abril', 'Maio', 'Junho', 'Julho', 'Agosto', 'Setembro'
, 'Outubro', 'Novembro', 'Dezembro']
    n = 1

    while(n < 4):
        mes = int(input("Escolha um mês (1-12): "))
        if 1 <= mes < 13:
            print('O mês é {}'.format(meses[mes-1]))
        n += 1
```

E testamos:

```
>>> Escolha um mês (1-12): 4
O mês é Abril
>>> Escolha um mês (1-12): 11
O mês é Novembro
>>> Escolha um mês (1-12): 6
O mês é Junho
```

Primeiro criamos um lista, relacionamos seus índices com os meses do ano, recuperamos seus valores através da entrada do usuário e imprimimos na tela o mês escolhido (mes[mes-1]) indexado a lista a partir do zero. Usamos um laço `while` que faz nosso programa entrar em *loop* e ser rodado 3 vezes.

Além de acessar um valor específico utilizando o índice, podemos acessar múltiplos valores através do fatiamento. Também utilizamos colchetes para o fatiamento. Suponha que queremos acessar os dois primeiros elementos de uma lista:

```
>>> lista = [2, 3, 5, 7, 11]
>>> lista[0:2]
[2, 3]
```

Podemos ter o mesmo comportamento fazendo:

```
>>> lista[:2]
[2, 3]
```

Se queremos todos os valores excluindo os dois primeiros, fazemos:

```
>>> lista[2:]
[5, 7, 11]
```

Ou utilizamos índices negativos:

```
>>> lista[-3:]
[5, 7, 11]
```

Também podemos fatiar uma lista de modo a pegar elementos em um intervalo específico:

```
>>> lista[2:4]
[5, 7]
```

As listas também possuem funcionalidades prontas, e podemos manipulá-las através de funções embutidas. As listas podem ser utilizadas em conjunto com uma função chamada `append()`, que adiciona um dado na lista:

```
>>> lista = []
>>> lista.append('zero')
>>> lista.append('um')
>>> lista
['zero', 'um']
```

A função `append()` só consegue inserir um elemento por vez. Se quisermos inserir mais elementos, podemos somar ou multiplicar listas, ou então utilizar a função `extend()`:

```
>>> lista = ['zero', 'um']
>>> lista.extend(['dois', 'três',])
>>> lista += ['quatro', 'cinco']
>>> lista + ['seis']
['zero', 'um', 'dois', 'três', 'quatro', 'cinco', 'seis']
>>> lista * 2
['zero', 'um', 'dois', 'três', 'quatro', 'cinco', 'seis', 'zero', 'um', 'dois', 'três', 'quatro',
'cinco', 'seis']
```

Isso é possível pois listas são sequências **mutáveis**, ou seja, conseguimos adicionar, remover e modificar seus elementos. Para imprimir o conteúdo de uma lista, podemos utilizar o comando `for`:

```
for valor in lista:
... print(valor)
...
zero
um
dois
três
quatro
cinco
```

- **Tuplas**

Uma tupla é uma lista **imutável**, ou seja, uma tupla é uma sequência que não pode ser alterada depois de criada. Uma tupla é definida de forma parecida com uma lista com a diferença do delimitador. Enquanto listas utilizam colchetes como delimitadores, as tuplas usam parênteses:

```
>>> dias = ('domingo', 'segunda', 'terça', 'quarta', 'quinta', 'sexta', 'sabado')
>>> type(dias)
```

```
<class 'tuple'>
```

Podemos omitir os parênteses e inserir os elementos separados por vírgula:

```
>>> dias = 'domingo', 'segunda', 'terça', 'quarta', 'quinta', 'sexta', 'sabado'
>>> type(dias)
>>> <class 'tuple'>
```

Note que, na verdade, é a vírgula que faz uma tupla, não os parênteses. Os parênteses são opcionais, exceto no caso da tupla vazia, ou quando são necessários para evitar ambiguidade sintática.

Assim como as listas, também podemos usar uma função para criar uma tupla passando um tipo que pode ser iterável como uma *string* ou uma lista. Essa função é a `tuple()`:

```
>>> texto = 'python'
>>> tuple(texto)
('p', 'y', 't', 'h', 'o', 'n')
>>> lista = [1, 2, 3, 4]
>>> tuple(lista)
(1, 2, 3, 4)
```

As regras para os índices são as mesmas das listas, exceto para elementos também imutáveis. Como são imutáveis, uma vez criadas não podemos adicionar nem remover elementos de uma tupla. O método `append()` da lista não existe na tupla:

```
>>> dias.append('sabado2')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>>
>>> dias[0]
'domingo'
>>>
>>> dias[0] = 'dom'
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Não é possível atribuir valores aos itens individuais de uma tupla, no entanto, é possível criar tuplas que contenham objetos mutáveis, como listas.

```
>>> lista = [3, 4]
>>> tupla = (1, 2, lista)
>>> tupla
(1, 2, [3, 4])
>>> lista = [4, 4]
>>> tupla
(1, 2, [4, 4])
>>> tupla[2] = [3, 4]
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

As tuplas são imutáveis e geralmente contêm uma sequência heterogênea de elementos. Já as listas são mutáveis, e seus elementos geralmente são homogêneos, sendo acessados pela iteração da lista, embora não seja uma regra.

Quando é necessário armazenar uma coleção de dados que não possa ser alterada, prefira usar tuplas a listas. Outra vantagem é que tuplas podem ser usadas como chaves de dicionários, que discutiremos nas seções adiante.

Tuplas são frequentemente usadas em programas Python. Um uso bastante comum são em funções que recebem múltiplos valores. As tuplas implementam todas as operações de sequência comuns.

- **Range**

O range é um tipo de sequência imutável de números, sendo comumente usado para *looping* de um número específico de vezes em um comando `for` já que representam um intervalo. O comando **range** gera um valor contendo números inteiros sequenciais, obedecendo a sintaxe:

```
range(inicio, fim)
```

O número finalizador, o *fim*, não é incluído na sequência. Vejamos um exemplo:

```
>>> sequencia = range(1, 3)
>>> print(sequencia)
range(1, 3)
```

O range não imprime os elementos da sequência, ele apenas armazena seu início e seu final. Para imprimir seus elementos precisamos de um laço `for` :

```
>>> for valor in range(1, 3):
...     print(valor)
...
1
2
```

Observe que ele não inclui o segundo parâmetro da função range na sequência. Outra característica deste comando é a de poder controlar o passo da sequência adicionando um terceiro parâmetro, isto é, a variação entre um número e o seu sucessor:

```
>>> for valor in range(1, 10, 2):
...     print(valor)
...
1
3
5
7
9
```

Os intervalos implementam todas as operações de seqüência comuns, exceto concatenação e repetição (devido ao fato de que objetos de intervalo só podem representar sequências que seguem um padrão estrito e a repetição e a concatenação geralmente violam esse padrão).

## 5.3 CONJUNTOS

O Python também inclui um tipo de dados para conjuntos. Um conjunto, diferente de uma sequência,

é uma coleção **não ordenada** e que **não admite elementos duplicados**.

Chaves ou a função `set()` podem ser usados para criar conjuntos.

```
>>> frutas = {'laranja', 'banana', 'uva', 'pera', 'laranja', 'uva', 'abacate'}
>>> frutas
>>> {'uva', 'abacate', 'pera', 'banana', 'laranja'}
>>> type(frutas)
<class 'set'>
```

Usos básicos incluem testes de associação e eliminação de entradas duplicadas. Os objetos de conjunto também suportam operações matemáticas como união, interseção, diferença e diferença simétrica. Podemos transformar um texto em um conjunto com a frunção `set()` e testar os operações:

```
>>> a =  set('abacate')
>>> b = set('abacaxi')
>>> a
{'a', 'e', 'c', 't', 'b'}
>>> b
{'a', 'x', 'i', 'c', 'b'}
>>> a - b                              # diferença
{'e', 't'}
>>> a | b                              # união
{'c', 'b', 'i', 't', 'x', 'e', 'a'}
>>> a & b                              # interseção
{'a', 'c', 'b'}
>>> a ^ b                              # diferença simétrica
{'i', 't', 'x', 'e'}
```

Note que para criar um conjunto vazio você tem que usar `set()`, não `{}`; o segundo cria um dicionário vazio, uma estrutura de dados que discutiremos na próxima seção.

```
>>> a = set()
>>> a
set()
>>> b = {}
>>> b
{}
>>> type(a)
<class 'set'>
>>> type(b)
<class 'dict'>
```

## 5.4 DICIONÁRIOS

Vimos que `list`, `tuple`, `range` e `str` são sequências ordenadas de objetos, e *sets* são coleções de elementos não ordenados. Dicionário é outra estrutura de dados em Python e seus elementos, sendo estruturadas de forma não ordenada assim como os conjuntos. Ainda assim, essa não é a principal diferença com as listas. Os dicionários são estruturas poderosas e muito utilizadas, já que podemos acessar seus elementos através de chaves e não por sua posição. Em outras linguagens, este tipo é conhecido como "matrizes associativas".

Qualquer chave de um dicionário é associada (ou mapeada) a um valor. Os valores podem ser

qualquer tipo de dado do Python. Portanto, os dicionários são pares de chave-valor não ordenados.

Os dicionários pertencem ao tipo de mapeamento integrado e não sequenciais como as listas, tuplas e *strings*. Vamos ver como isso funciona no código e criar um dicionário com dados de uma pessoa:

```
>>> pessoa = {'nome': 'João', 'idade': 25, 'cidade': 'São Paulo'}
>>> pessoa
'nome': 'João', 'idade': 25, 'cidade': 'São Paulo'
```

Os dicionários são delimitados por chaves ({}) e suas chaves ('nome', 'idade' e 'cidade') por aspas. Já os valores podem ser de qualquer tipo. No exemplo acima, temos duas *strings* e um `int` .

O que será que acontece se tentarmos acessar seu primeiro elemento?

```
>>> pessoa[0]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 0
```

Não é possível acessar um elemento de um dicionário por um índice como na lista. Devemos acessá-los por sua chave:

```
>>> pessoa['nome']
'João'
>>> pessoa['idade']
25
```

Se precisarmos adicionar algum elemento, como por exemplo, o país, basta fazermos:

```
>>> pessoa1['país'] = 'Brasil'
>>> pessoa1
{'nome': 'João', 'idade': 25, 'cidade': 'São Paulo', 'país': 'Brasil'}
```

Como sempre acessamos seus elementos através de chaves, o dicionário possui um método chamado `keys()` que devolve o conjunto de suas chaves:

```
>>> pessoa1.keys()
dict_keys(['nome', 'idade', 'cidade', 'pais'])
```

Assim como um método chamado `values()` que retorna seus valores:

```
>>> pessoa1.values()
dict_values(['João', 25, 'São Paulo', 'Brasil'])
```

Note que as chaves de um dicionário não podem ser iguais para não causar conflito. Além disso, somente tipos de dados imutáveis podem ser usados como chaves, ou seja, nenhuma lista ou dicionário pode ser usado. Caso isso aconteça, recebemos um erro:

```
>>> dic = {[1, 2, 3]: 'valor'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Já as tuplas, como chaves, são permitidas:

```
>>> dic = {(1, 2, 3): 'valor'}
>>> dic
{(1, 2, 3): 'valor'}
```

Também podemos criar dicionários utilizando a função **dict()**:

```
>>> a = dict(um=1, dois=2, três=3)
>>> a
{'três': 3, 'dois': 2, 'um': 1}
```



## 5.5 EXERCÍCIOS: ESTRUTURA DE DADOS

Vamos tentar resolver alguns desafios. Dada a lista = [12, -2, 4, 8, 29, 45, 78, 36, -17, 2, 12, 8, 3, 3, -52] faça um programa que:

```
a) imprima o maior elemento

b) imprima o menor elemento

c) imprima os números pares

d) imprima o número de ocorrências do primeiro elemento da lista

e) imprima a média dos elementos

f) imprima a soma dos elementos de valor negativo
```

1. Primeiramente, vamos gerar um novo arquivo para este código, chamado `lista.py`. Crie o arquivo no Pycharm dentro do mesmo projeto 'jogos' criado no capítulo anterior.

2. Vamos começar gerando um loop para percorrer a lista. Utilizamos um `for` junto com um `range` para percorrer cada índice da nossa lista:

```
lista = [12, -2, 4, 8, 29, 45, 78, 36, -17, 2, 12, 12, 3, 3, -52]

for index in range(0, len(lista)):
```

3. Agora, vamos resolver o item a. Defina uma variável fora do seu for chamada `maiorValor` e a iguale ao primeiro elemento na lista. Dentro do seu `for`, percorra os elementos dentro de um `if` para substituir o valor encontrado caso seja maior do que o mesmo:

```
lista = [12, -2, 4, 8, 29, 45, 78, 36, -17, 2, 12, 12, 3, 3, -52]

maiorValor = lista[0]

for index in range(0, len(lista)):
    #Maior valor
    if(maiorValor < lista[index]):
        maiorValor = lista[index]

print(maiorValor)
```

4. Para resolver o item `b`, basta seguir a mesma ideia do exemplo anterior. Crie um outro `if` abaixo do que você criou no passo anterior, apenas mudando o operador da condição:

```
menorValor = lista[0]

for index in range(0, len(lista)):
#... seu código aqui
#Menor valor
if(menorValor > lista[index]):
    menorValor = lista[index]

print(menorValor)
```

5. Para resolver o item `c`, basta definir uma lista, e caso o valor atual da lista com módulo 2 retorne 0, ele é adicionado na mesma:

```
listaPares = []

for index in range(0, len(lista)):
    #... seu código aqui
    #Numeros pares
    if( lista[index] % 2 == 0):
        listaPares.append(lista[index])
    print(listaPares)
```

6. Para resolver o item `d`, é preciso verificar se o item atual da lista a ser percorrida coincide com o elemento em seu primeiro índice:

```
ocorrenciasItem1 = 0

for index in range(0, len(lista)):
    #... seu código aqui
    #Numero de ocorrencias
    if(lista[index] == lista[0]):
        ocorrenciasItem1 = ocorrenciasItem1 + 1

print(ocorrenciasItem1)
```

7. A resolução do item `e` requer a implementação dentro e fora do `for`. Dentro do `for`, some cada um dos elementos em uma variável única. Após o `loop`, divida o valor obtido pelo total de elementos na sua lista:

```
mediaElementos = 0

for index in range(0, len(lista)):
    #... seu código aqui
    #Media de elementos
    mediaElementos =+ mediaElementos + lista[index]
mediaElementos = mediaElementos / len(lista)

print(mediaElementos)
```

8. Por fim, para resolver o item `f`, faça a soma de todos os números negativos somando todos os valores que são menores que 0:

```
somaNegativos = 0

for index in range(0, len(lista)):
    #... seu código aqui
    #Soma dos números negativos
    if(lista[index] < 0):
        somaNegativos = somaNegativos + lista[index]

print(somaNegativos)
```

9. Tente imprimir todas estas condições no seu `loop` e veja o resultado. O seu resultado deverá estar similar a:

```python
lista = [12, -2, 4, 8, 29, 45, 78, 36, -17, 2, 12, 12, 3, 3, -52]

#declarando nossas variáveis
maiorValor = lista[0]
menorValor = lista[0]
listaPares = []
ocorrenciasItem1 = 0
mediaElementos = 0
somaNegativos = 0

#iniciando o nosso loop:
for index in range(0, len(lista)):

    #Maior valor
    if(maiorValor < lista[index]):
        maiorValor = lista[index]

    #Menor valor
    if(menorValor > lista[index]):
        menorValor = lista[index]

    #Numeros pares
    if(lista[index] % 2 == 0):
        listaPares.append(lista[index])

    #Numero de ocorrências
    if(lista[index] == lista[0]):
        ocorrenciasItem1 = ocorrenciasItem1 + 1

    #Soma dos números negativos
    if(lista[index] < 0):
        somaNegativos = somaNegativos + lista[index]
```

```
        #Media do somatório dos elementos
        mediaElementos =+ mediaElementos + lista[index]


    mediaElementos = mediaElementos / len(lista)

    print("Maior valor: " + str(maiorValor))
    print("Menor valor: " + str(menorValor))
    print("Lista de elementos pares: " + str(listaPares))
    print("Número de ocorrências do primeiro item: " + str(ocorrenciasItem1))
    print("Média dos elementos: " + str(mediaElementos))
    print("Somatório dos valores negativos:" + str(somaNegativos))
```