Macetes em Python

1. ## Troca de Variáveis:

   Troque os valores de duas variáveis sem uma variável temporária.

   ```
   a, b = 1, 2
   a, b = b, a
   print(a, b)
   ```

   Saída Esperada:

   ```
   2 1
   ```

2. ## List Comprehension:

   Crie listas de forma concisa e eficiente.

   ```
   squares = [x**2 for x in range(5)]
   print(squares)
   ```

   Saída Esperada:

   ```
   [0, 1, 4, 9, 16]
   ```

3. ## Unpacking de Listas e Tuplas:

   Desempacote elementos diretamente em variáveis.

   ```
   a, b, c = [1, 2, 3]
   print(a, b, c)
   ```

   Saída Esperada:

   ```
   1 2 3
   ```

4. ## Verificar Substrings em Strings:

   Use `in` para verificar se uma substring está em uma string.

   ```
   if "py" in "python":
       print("Found!")
   ```

   Saída Esperada:

   ```
   Found!
   ```

5. ## Concatenar Strings:

Use `join()` para concatenar strings em uma lista.

```
words = ["Hello", "World"]
sentence = " ".join(words)
print(sentence)
```

Saída Esperada:

```
Hello World
```

6. ## Dicionário com Valores Padrão:

Use `defaultdict` do módulo `collections` para evitar `KeyError`.

```
from collections import defaultdict
d = defaultdict(int)
d["key"] += 1
print(d["key"])
```

Saída Esperada:

```
1
```

7. ## Expressões Ternárias:

Simplifique declarações `if-else` em uma única linha.

```
x = 4
result = "Even" if x % 2 == 0 else "Odd"
print(result)
```

Saída Esperada:

```
Even
```

8. ## Iterando com Enumerate:

Use `enumerate()` para obter o índice e o valor ao iterar sobre uma lista.

```
for index, value in enumerate(["a", "b", "c"]):
    print(index, value)
```

Saída Esperada:

```
0 a
1 b
2 c
```

9. ## Contando Elementos em uma Lista:

Use `Counter` do módulo `collections` para contar a frequência de elementos.

```
from collections import Counter
counts = Counter([1, 2, 2, 3, 3, 3])
print(counts)
```

Saída Esperada:

```
Counter({3: 3, 2: 2, 1: 1})
```

10. ## Função Lambda:

Use funções anônimas para tarefas rápidas.

```
square = lambda x: x ** 2
print(square(5))
```

Saída Esperada:

```
25
```

11. ## Operador de Desempacotamento `*`:

Use `*` para desempacotar listas ou tuplas ao passar para funções.

```
def add(x, y):
    return x + y

numbers = [1, 2]
result = add(*numbers)
print(result)
```

Saída Esperada:

```
3
```

12. ## Operador de Desempacotamento de Dicionários `**`:

Use `**` para passar argumentos de um dicionário para uma função.

```
def greet(name, age):
    return f"Hello, {name}. You are {age} years old."

person = {"name": "Alice", "age": 30}
message = greet(**person)
print(message)
```

Saída Esperada:

```
Hello, Alice. You are 30 years old.
```

13. ## Manuseio de Arquivos Contextualizado:

Use `with` para manusear arquivos sem se preocupar com o fechamento explícito.

```
with open("file.txt", "w") as file:
    file.write("Hello, World!")

with open("file.txt", "r") as file:
    data = file.read()
print(data)
```

Saída Esperada:

```
Hello, World!
```

14. Geradores com `yield`:

Use `yield` para criar geradores que economizam memória.

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for number in countdown(5):
    print(number)
```

Saída Esperada:

```
5
4
3
2
1
```

15. Funções Decoradoras:

Use decoradores para adicionar funcionalidades extras às funções.

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Saída Esperada:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```