Aula 19

Revisão

Empacotando e Desempacotando

O empacotamento se refere à criação de uma única variável que contém vários valores. Isso é feito por meio de tuplas ou listas em Python.

Já o desempacotamento se refere à extração de valores individuais de uma variável empacotada.

Abaixo temos o exemplo do uso com uma lista.

```
>>> # empacotando
>>> lista = ['asdf', 42, True, [3,5], 3.14, (9,0)]
>>> print(lista)
['asdf', 42, True, [3, 5], 3.14, (9, 0)]
>>>
>>> # desempacotando
>>> print(*lista)
asdf 42 True [3, 5] 3.14 (9, 0)
>>>
```

*args e **kwargs nos permitem realizar o empacotamento e desempacotamento nas funções.

O parâmetro *args é usado para passar um número variável de argumentos posicionais para uma função.

O parâmetro **kwargs é usado para passar um número variável de argumentos nomeados para uma função.

Quando um parâmetro é precedido por dois asteriscos (**), isso significa que ele receberá todos os argumentos nomeados restantes como um dicionário.

Abaixo temos um exemplo dos dois conceitos atuando juntos.

```
>>> def combinado(*args, **kwargs):
...     """Função combinados."""
...     print(f'Valor args = {args}.')
...     print(f'Valor kwargs = {kwargs}.\n')
...     for arg in args:
...             print(f'Valor arg {arg}.')
...     for c, v in kwargs.items():
...             print(f'{c} : {v}.')
...
>>> combinado(1,2,3,42,nome='fulano',idade=42)
Valor args = (1, 2, 3, 42).
Valor kwargs = {'nome': 'fulano', 'idade': 42}.

Valor arg 1.
Valor arg 2.
Valor arg 3.
Valor arg 42.
nome : fulano.
idade : 42.
>>>
```

Agora, um exemplo da combinação do que vimos até agora sobre os parâmetros e argumentos das funções.

```
>>> def tudo(a, b=42, *args, **kwargs):
...     """Função demonstrando o uso de todos os parâmetros possíveis"""
...     print(f"a={a}")
...     print(f"b={b}")
...     print(f"args={args}")
...     print(f"kwargs={kwargs}")
...
>>>
```

```
>>> tudo(1)
a=1
b=42
args=()
kwargs={}
>>>
```

```
>>> tudo(1, 2)
a=1
b=2
args=()
kwargs={}
>>>
```

```
>>> tudo(1, 2, 3, 4, 5)
a=1
b=2
args=(3, 4, 5)
kwargs={}
>>>
```

```
>>> tudo(1, 2, 3, 4, 5, nome="Fulano", idade=42)
a=1
b=2
args=(3, 4, 5)
kwargs={'nome': 'Fulano', 'idade': 42}
>>>
```

No exemplo acima, temos todos os conceitos aplicados em apenas uma função.

Recursão

A recursão é uma técnica de programação em que uma função chama a si mesma para resolver um problema de forma iterativa. Em outras palavras, a recursão é a repetição de um processo dentro de si mesmo até que uma condição de saída seja atendida.

Em Python, a recursão é implementada usando uma função que chama a si mesma com argumentos diferentes até que uma condição de saída seja atingida. A condição de saída é geralmente definida como uma base, que é uma condição em que a função recursiva para de chamar a si mesma e retorna um resultado final.

Pense na recursão como uma forma de realizar um loop de repetição, mas usando funções.

```
>>> def somador(n):
...     if n == 1:
...         return n
...     return n + somador(n - 1)
...
>>> print(f'A soma dos números de 4 até 0 é {somador(4)}.')
A soma dos números de 4 até 0 é 10.
>>> print(f'A soma dos números de 5 até 0 é {somador(5)}.')
A soma dos números de 5 até 0 é 15.
>>> print(f'A soma dos números de 15 até 0 é {somador(15)}.')
A soma dos números de 15 até 0 é 120.
>>>
```

No exemplo acima, a função somador é chamada passando o valor 4 (na primeira chamada). Dentro da função, é feito um teste se o valor de n é igual a 1. Se for, retorna o valor de n que é 1. Se não for, pega o valor de n e soma com uma nova chamada da função somador, mas agora decrementando o valor passado para 3 (n – 1). Então, todo o processo é repetido, até que n atinja o valor de 1.

Abaixo, temos um exemplo do uso do cálculo de fatorial com e sem recursão.

```
>>> def fatorial_while(n):
...     total = 1
...     while n > 0:
...         total = total * n
...         n -= 1
...     return total
...
>>> def fatorial_recursao(n):
...     if n == 0:
...         return 1
...     return n * fatorial_recursao(n-1)
...
>>> print(f'Fatorial com loop {fatorial_while(5)}.')
Fatorial com loop 120.
>>>
>>> print(f'Fatorial com recursão {fatorial_recursao(5)}.')
Fatorial com recursão 120.
>>>
```

Assim fica fácil ver as diferenças e como o fatorial com a recursão é mais elegante.
A função fatorial_recursao() calcula o fatorial de um número inteiro usando a recursão. A condição de saída é quando n == 0, momento em que a função retorna 1. Caso contrário, a função chama a si mesma com n-1 como argumento e multiplica o resultado pelo valor de n.
Ao chamar a função fatorial_recursao(5), no exemplo, a função é chamada com n igual a 5. Como n não é igual a 0, a função chama a si mesma com n-1, que é igual a 4. A função continua a se chamar recursivamente até que n seja igual a 0, momento em que a condição de saída é atingida e o valor final é retornado.

A recursão pode ser uma técnica poderosa de programação, mas é importante ter cuidado ao usá-la para evitar loops infinitos e esgotamento de recursos do sistema. Além disso, em alguns casos, a recursão pode ser menos eficiente do que a implementação iterativa de uma solução de problema.

Contagem regressiva.

```
>>> def contagem(n):
...     if n == 0:
...         print('Lançamento!')
...     else:
...         print(f'Contagem em {n}.')
...         contagem(n-1)
...
>>> contagem(10)
Contagem em 10.
Contagem em 9.
Contagem em 8.
Contagem em 7.
Contagem em 6.
Contagem em 5.
Contagem em 4.
Contagem em 3.
Contagem em 2.
Contagem em 1.
Lançamento!
>>>
```

Cálculo de Fibonacci usando recursão

```
>>> def fibonacci(n):
...     if n ≤ 1:
...         return n
...     return fibonacci(n-1) + fibonacci(n-2)
...
>>> print(f'Fibonacci 5 {fibonacci(5)}.')
Fibonacci 5 5.
>>> print(f'Fibonacci 10 é {fibonacci(10)}.')
Fibonacci 10 é 55.
>>> print(f'Fibonacci 20 é {fibonacci(20)}.')
Fibonacci 20 é 6765.
>>>
```

Neste exemplo, a função fibonacci() chama a si mesma para calcular os números de Fibonacci até que a condição de saída n <= 1 seja atingida. Quando n é igual a 0 ou 1, a função retorna o valor de n. Caso contrário, a função chama a si mesma com n-1 e n-2 como argumentos e retorna a soma dos resultados.

The Zen of Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Tradução:

Bonito é melhor do que feio.
Explícito é melhor do que implícito.
Simples é melhor do que complexo.
Complexo é melhor do que complicado.
Linear é melhor do que aninhado.
Esparso é melhor do que denso.
Legibilidade conta.
Casos especiais não são especiais o suficiente para quebrar as regras.
Ainda que praticidade vença a pureza.
Erros nunca devem passar silenciosamente.
A menos que sejam explicitamente silenciados.
Diante da ambiguidade, recuse a tentação de adivinhar.
Deveria haver preferencialmente um - e preferencialmente apenas um - modo óbvio de fazer algo.
Embora esse modo possa não ser óbvio a princípio, a menos que você seja holandês.
Agora é melhor do que nunca.
Embora nunca seja melhor que agora mesmo.
Se a implementação é difícil de explicar, é uma má ideia.
Se a implementação é fácil de explicar, pode ser uma boa ideia.
Namespaces são uma grande ideia - vamos ter mais dessas!

Bônus: https://www.youtube.com/watch?v=hgI0p1zf31k

Exercícios para Praticar

1. Escreva uma função recursiva que imprima os números inteiros de 1 a n.
2. Escreva uma função recursiva que calcule a soma dos números inteiros de 1 a n.
3. Escreva uma função recursiva que calcule o fatorial de um número inteiro n.
4. Escreva uma função recursiva que calcule o enésimo número na sequência de Fibonacci.
5. Escreva uma função recursiva que determine se um número inteiro é primo.
6. Escreva uma função recursiva que calcule a potência de um número inteiro x elevado a um expoente inteiro n.
7. Escreva uma função recursiva que determine se uma string é um palíndromo.
8. Escreva uma função recursiva que inverta uma string.
9. Escreva uma função recursiva que determine o máximo divisor comum entre dois números inteiros a e b.
10. Escreva uma função recursiva que calcule o mínimo múltiplo comum entre dois números inteiros a e b.
11. Escreva uma função recursiva que determine se uma lista de inteiros está em ordem crescente.
12. Escreva uma função recursiva que calcule a soma de uma lista de inteiros.
13. Escreva uma função recursiva que calcule o produto de uma lista de inteiros.
14. Escreva uma função recursiva que calcule o número de ocorrências de um elemento em uma lista.
15. Escreva uma função recursiva que determine se um elemento está em uma lista.
16. Escreva uma função recursiva que inverta uma lista.
17. Escreva uma função recursiva que determine se uma lista é um palíndromo.
18. Escreva uma função recursiva que calcule o máximo valor em uma lista de inteiros.
19. Escreva uma função recursiva que calcule o mínimo valor em uma lista de inteiros.
20. Escreva uma função recursiva que calcule a mediana de uma lista de inteiros.
21. Escreva uma função recursiva que determine se uma palavra é formada por uma combinação de outras duas palavras, em ordem.
22. Escreva uma função recursiva que calcule a média de uma lista de números inteiros.
23. Escreva uma função recursiva que determine se um número inteiro é par ou ímpar.
24. Escreva uma função recursiva que calcule a soma de todos os elementos em uma matriz de inteiros.
25. Escreva uma função recursiva que calcule o produto de todos os elementos em uma matriz de inteiros.
26. Escreva uma função recursiva que calcule o determinante de uma matriz quadrada de inteiros.
27. Escreva uma função recursiva que calcule a raiz quadrada de um número real positivo.
28. Escreva uma função recursiva que calcule o número de maneiras diferentes de escolher k elementos de um conjunto com n elementos.
29. Escreva uma função recursiva que calcule o número de maneiras diferentes de distribuir n objetos idênticos em k recipientes distintos.
30. Escreva uma função recursiva que determine se uma lista de inteiros contém três números consecutivos em ordem crescente.