Aula 12 - Funções

Não importa a linguagem de programação que você codifica: um dos principais recursos, se não for **O PRINCIPAL** recurso utilizado em programação, e que é praticamente **OBRIGATÓRIO** que qualquer programador em qualquer nível saiba, são as **Funções**.

Funções são pequenos blocos de programação que podem ser chamados no algoritmo principal. Isso possibilita que seu código seja dividido em partes menores. Essa técnica visa atingir 5 objetivos principais:

- Facilitar a leitura do código-fonte.
- Otimizar o programa.
- Reutilizar o mesmo código sem necessidade de copiar e colar diversas vezes.
- Utilizar um determinado trecho de código somente quanto neccessário.
- Facilitar a manutenção do sistema.

Essas funções podem ser escritas dentro do mesmo arquivo ou em um arquivo separado, sendo chamado através das **importações** (veremos isso em aulas futuras). Para a aula de hoje, vamos trabalhar apenas com funções dentro do mesmo arquivo do código-fonte principal. Geralmente essas funções representam uma determinada funcionalidade ou recurso do programa que está sendo escrito (sabe as fórmulas do Excel? São exemplos de funções).

Por ser um bloco de programação, quase todo o código dentro da função precisa estar indentado, exceto a primeira linha, que contém o comando `def` seguido do nome da função. Para quem tem experiência com programação, já deve imaginar que `def` é o comando que cria uma função, ao invés do tradicional `function` de linguagens como o JavaScript e o PHP, por exemplo. A nomenclatura das funções segue um padrão similar ao das variáveis, com nomes o mais simples possível, em caixa baixa, sem utilizar palavras reservadas pelo sistema, sem espaço, nem acento e nem "ç", e também não pode começar com números. Caso o nome da função possua mais de uma palavra, utiliza-se a convenção **Snake Case**. Para não ocorrer confusões, uma prática interessante seguida por alguns programadores é dar o nome da função usando um verbo no infinitivo, pois uma função indica uma ação a ser executada. Exemplo: chamar, alterar, inserir, calcular, etc...

Outro detalhe é que o nome das funções sempre terminam com parênteses (`()`). Esses parênteses significam **parâmetros**, mas serão explicados melhor mais para frente, ainda nesta aula. Por hora, saiba que eles precisam ser colocados independentemente de estarem vazios ou não. Uma vez que a função é criada, ela pode ser chamada a qualquer momento no algoritmo principal, quantas vezes for necessário. Tudo o que precisa é apenas chamar o nome da função. Vamos para um exemplo mais prático. Logo abaixo temos um exemplo de uma função que exibe uma mensagem de boas vindas na tela:

In [ ]:
```python
# função que exibe uma mensagem de boas vindas
def exibir_msg():
    print('-'*10,'BEM VINDO AO SENAI', '-'*10)
    print('Você está fazendo o CURSO DESENVOLVEDOR PYTHON!')

# programa principal
decisao = input('Gostaria de exibir a mensagem? "sim" para exibir ou "não" para

match decisao:
    case 'sim':
        exibir_msg()
    case 'não':
        pass
    case _:
        print('Resposta inválida!')
```

```
---------- BEM VINDO AO SENAI ----------
Você está fazendo o CURSO DESENVOLVEDOR PYTHON!
```

Parâmetros

Mutas funções precisam receber dados de fora dela para funcionar. Esses dados são passados para as funções dentro dos parênteses que ficam no final do nome da função. São os **parâmetros**. Os parâmetros são variáveis que são declaradas dentro do parênteses durante a criação das funções, e seus valores são repassados para elas durante a chamada da função no algoritmo principal. Funciona assim: `def nome_da_funcao(variavel):` na criação da função, e `nome_da_funcao(valor_da_varivavel)` na chamada da função. Segue o exemplo abaixo:

In [ ]:
```python
# função que recebe o nome do usuário e envia uma mensagem personalizada
def exibir_msg(nome):
    print(f'{'-'*20} Olá {nome}, seja bem vindo ao SENAI {'-'*20}')

# programa principal
nome = input('Informe seu nome: ')

# chama a função
exibir_msg(nome)
```

```
-------------------- Olá Alex Machado, seja bem vindo ao SENAI ------------------
--
```

Uma função pode receber nenhum, um, ou mais de um parâmetros, quantos a função precisar. Vamos repetir o último exemplo, mas desta vez recebendo 2 parâmetros:

In [ ]:
```python
# função que recebe o nome do usuário e do curso, e envia uma mensagem personali
def exibir_msg(nome, curso):
    print(f'{'-'*20} Olá {nome}, seja bem vindo ao SENAI {'-'*20}')
    print(f'Você está fazendo o curso {curso}.')

# programa principal
nome = input('Informe seu nome: ')
curso = input('Informe o nome do seu curso: ')
```

```
# chama a função
exibir_msg(nome, curso)
```

```
-------------------- Olá Alex, seja bem vindo ao SENAI --------------------
Você está fazendo o curso Desenvolvedor Python.
```

Retorno

Algumas funções podem retornar um valor para ser usado no algoritmo principal. Isso é muito útil para repassar o valor da função para uma saída de dados ou para uma variável, por exemplo. Uma função pode ou não ter retorno. Em outras linguagens de programação, como o Pascal ou Delphi, chamamos as funções que **não** possuem retorno de **Procedimentos**, ou ***Procedures***.

Return

O comando `return` é usado em funções regulares para retornar um valor e encerrar a execução da função. Quando o `return` é atingido, ela retorna um valor e para a execução, encerrando a função.

No exemplo abaixo, vamos criar uma função que calcula a equação do 1º grau a partir de 2 valores, e retorna o resultado. Veja:

In [ ]:
```
# função que calcula a equação do 1º grau
def calcular_equacao_primeiro_grau(a, b):
    x = -b/a
    return x

# programa principal
a = int(input('Informe o valor de "a": '))
b = int(input('Informe o valor de "b": '))

print(f'O valor de "x" na equação do 1º grau é {calcular_equacao_primeiro_grau(a
```

```
O valor de "x" na equação do 1º grau é 0.5.
```

A função só pode ter um único `return` apenas. A "exceção", se é que se pode dizer assim, é quando esse retorno está embutido dentro de uma estrutura de decisão. Sendo assim, a função pode retornar um ou outro valor. Vejamos no exemplo abaixo, onde a função decide se uma pessoa está matriculada em um curso ou não com base na validação da documentação:

In [ ]:
```
# função
def matricular_aluno(nome, validacao):
    if validacao == True:
        return f'{nome} foi matriculado com sucesso!'
    else:
        return f'{nome} teve a matrícula indeferida!'

# programa principal
nome = input('Informe o nome do aluno: ')
documentacao = input('A documentação foi entregue? ')
```

```
if documentacao == 'sim':
    validacao = True
else:
    validacao = False

# chama a função
print(matricular_aluno(nome, validacao))
```

```
Alex foi matriculado com sucesso!
```

Na verdade, há ainda um outro caso em que há uma "exceção" sobre a função retornar mais de um valor, mas não é com o comando `return` ...

Yield

O comando `yield` é um tipo de retorno de função usado em funções geradoras para retornar valores de forma **iterativa**, sem encerrar a execução da função. A função mantém seu estado entre as chamadas e continua de onde parou quando é chamada novamente, retornando o próximo valor. Para isso, a função é chamada dentro do comando `next()`. Veja abaixo:

In [ ]:
```
# função
def verificar_matricula(nome):
    yield f'{nome} está com a documentação pendente.'
    yield f'{nome} está com a documentação em fase de verificação.'
    yield f'{nome} está com a documentação aprovada e com a matrícula pendente.'
    yield f'{nome} está com a matrícula aprovada e efetivada.'

# programa principal
nome = input('Informe o nome do aluno que deseja verificar a situação da matrícu

# o retorno da função é repassada para uma variável
situacao = verificar_matricula(nome)

# saída de dados: o mesmo comando vai mostrar 3 saídas diferentes
print(next(situacao))
print(next(situacao))
print(next(situacao))
print(next(situacao))
```

```
Alex está com a documentação pendente.
Alex está com a documentação em fase de verificação.
Alex está com a documentação aprovada e com a matrícula pendente.
Alex está com a matrícula aprovada e efetivada.
```

Outra possibilidade é de exibir o retorno da função em um laço de repetição. Nesse caso, não haveria necessidade do comando `next()`. Veja abaixo o mesmo programa sendo executado com um laço `for`:

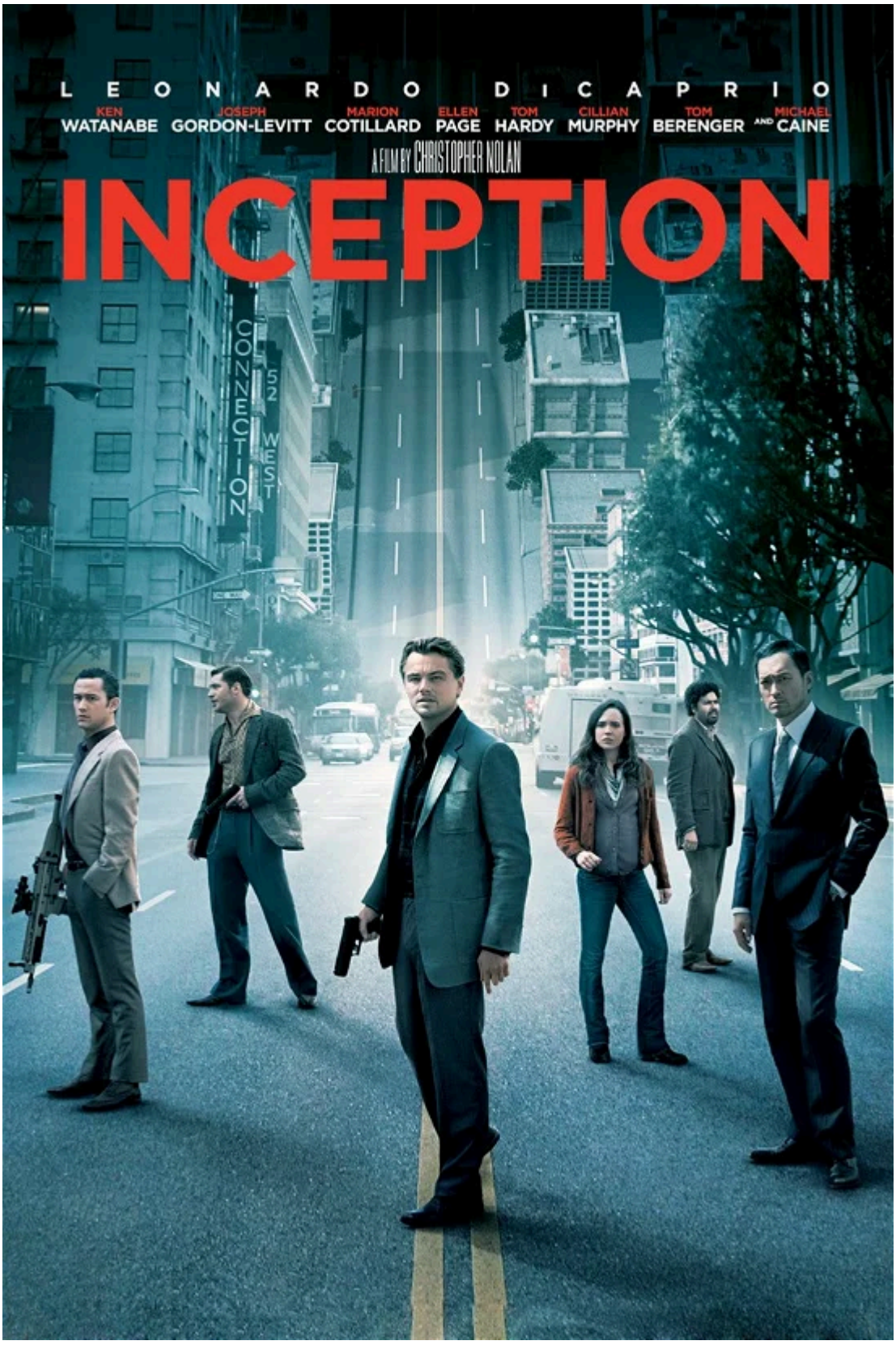
In [ ]:
```
# função
def verificar_matricula(nome):
    yield f'{nome} está com a documentação pendente.'
    yield f'{nome} está com a documentação em fase de verificação.'
    yield f'{nome} está com a documentação aprovada e com a matrícula pendente.'
    yield f'{nome} está com a matrícula aprovada e efetivada.'
```

```
# programa principal
nome = input('Informe o nome do aluno que deseja verificar a situação da matrícu

# o retorno da função é repassada para uma variável
situacoes = verificar_matricula(nome)

# saída de dados exibida em um laço for
for situacao in situacoes:
    print(situacao)
```

```
Alex Machado está com a documentação pendente.
Alex Machado está com a documentação em fase de verificação.
Alex Machado está com a documentação aprovada e com a matrícula pendente.
Alex Machado está com a matrícula aprovada e efetivada.
```

Função recursiva

Você talvez já deve ter visto o filme **A Origem** (no original ***Inception***), dirigido por Christopher Nolan, com Leonardo DiCarpio, onde o protagonista precisa entrar em um sonho dentro de outro sonho...dentro de outro sonho para implementar uma ideia na mente de um dos personagens do filme (o que ele consegue só depois de entrar no limbo, que é um sonho dentro do outro sonho dentro do outro sonho dentro do sonho original). O que acontece lá pode ser descrito no mundo da programação como uma **Recursão**, ou ainda **Recursividade**.



Filme *Inception*

Uma função recursiva é nada mais do que uma função que chama a si mesma quando executada. Assim, toda vez que essa função é executada, acaba sendo executada por ela mesma de novo, e uma vez que ela é executada por ela mesma, acaba executando ela mesma mais uma vez, e de novo, e de novo, e de novo....uma dentro da outra. Ela é utilizada para resolver problemas que podem ser divididos em subproblemas menores e

idênticos ao problema original. A recursão é uma técnica poderosa que permite a resolução elegante de problemas complexos.

As funções recursivas são extremamente difíceis de entender e de aplicar, já que elas devem convergir para o caso base em algum momento, caso contrário, pode acabar executando um ***loop infinito***, consumindo 100% da CPU e da RAM, e travando seu PC. Portanto, é essencial que a função seja projetada de forma cuidadosa, garantindo que a recursão seja encerrada corretamente.

Olha a recursão aí na imagem.....

Vantagens

- **Clareza e legibilidade:** em alguns casos, a implementação de um algoritmo recursivo pode ser mais clara e legível do que uma solução iterativa. Isso ocorre especialmente quando o problema pode ser naturalmente dividido em subproblemas menores.
- **Solução elegante:** a recursão permite a resolução de problemas complexos de forma elegante, utilizando a própria definição do problema para resolvê-lo.
- **Reutilização de código:** a função recursiva em Python pode ser reutilizada em diferentes contextos, desde que o problema em questão possa ser dividido em subproblemas menores.

Desvantagens

- **Consumo de recursos:** a recursão pode consumir mais recursos do que uma solução iterativa, devido à pilha de chamadas que é criada a cada chamada recursiva. Isso pode levar a problemas de desempenho e até mesmo estourar a pilha de execução em casos extremos.
- **Dificuldade de depuração:** a depuração de funções recursivas pode ser mais complexa do que a depuração de soluções iterativas, pois é necessário acompanhar

o fluxo de execução em cada chamada recursiva.

- **Possibilidade de loop infinito:** se a função recursiva não for projetada corretamente, pode ocorrer um loop infinito, o que resultará em travamento do programa.

Apesar das desvantagens, o uso de funções recursivas em Python pode ser extremamente útil quando aplicado corretamente. Com a compreensão adequada do problema e a implementação cuidadosa da recursão, é possível criar algoritmos poderosos e eficientes.

Um excelente exemplo de uso de uma função recursiva é o cálculo de um fatorial. Veja o exemplo abaixo:

In [ ]:
```
# função recursiva que calcula o fatorial
def calcular_fatorial(n):
    if n == 0:
        return 1
    else:
        return n * calcular_fatorial(n-1)

# programa principal
n = int(input('Informe um número inteiro: '))
print(calcular_fatorial(n))
```

```
120
```

Um outro excelente exemplo é o cálculo de um determinado termo da sequência de *Fibonacci*. Veja abaixo:

In [ ]:
```
# função recursiva que calcula Fibonacci
def calcular_fibonacci(n):
    if n <= 1:
        return n
    else:
        return calcular_fibonacci(n-1) + calcular_fibonacci(n-2)

# programa principal
n = int(input('Informe o número de sequências a ser calculada: '))
print(calcular_fibonacci(n))
```

```
55
```

Inclusive, é uma boa oportunidade para aprender *Fibonacci*, um dos cálculos mais importantes da história da humanidade. Se ainda não sabe o que é, fica o link para um vídeo com o Pato Donald explicando o que é:

Link do vídeo: https://www.youtube.com/watch?v=XVLHX0ddtqo

Lambda

Há uma forma de se escrever funções de forma simplificada. Quando as funções são curtas (geralmente apenas com o retorno), você pode trocar por um ***lambda***, que nada

mais são do que funções sem nome, que já são atribuidas de forma direta às variáveis. Veja o exemplo abaixo. Ele representa um algoritmo com uma função normal:

In [ ]:
```
# função normal
def somar(x, y):
    result = x + y
    return result

# programa principal
x = int(input('Informe um número: '))
y = int(input('Informe outro número: '))

print(f'O resultado da soma é {somar(x, y)}.')
```

```
O resultado da soma é 3.
```

Veja o mesmo programa reescrito utilizando ***lambda*** no lugar da função:

In [ ]:
```
# lambda
soma = lambda x, y: x + y

# programa principal
x = int(input('Informe um número: '))
y = int(input('Informe outro número: '))

print(f'O resultado da soma é {soma(x, y)}.')
```

```
O resultado da soma é 3.
```

Caso as funções sejam pequenas e possuam retorno, convém trocar por lambda sempre que possível, para deixar seu código mais limpo e eficiente.

Função map()

Algumas aulas atrás, aprendemos sobre **coleções**. Agora, vamos aprender que podemos aplicar funções sobre alguns tipos de coleções, como uma lista, por exemplo. Isso pode ser feito tanto com uma função normal como com um lambda. O responsável por isso é a função `map()`. Sua estrutura é `map(função, lista)`. Como a saída é também uma lista, vamos usar também a função `list()` para exibir o resultado como uma lista.

No exemplo abaixo, vamos pegar uma lista numérica e calcular a **PG** dela (Progressão Geométrica). Para o nosso exemplo, usaremos um lambda:

In [ ]:
```
# lambda que calcula pg
calcular_pg = lambda x: x * 2

numeros = [1, 2, 3, 4, 5]
print(list(map(calcular_pg, numeros)))
```

```
[2, 4, 6, 8, 10]
```

Exercício

1. Crie uma aplicação de banco, onde o usuário se cadastra e cria uma conta corrente que começa com saldo de R$ 0,00. O usuário terá as opções: Criar conta, Exibir dados da conta, Depositar valor, Sacar valor, Encerrar conta, Sair do programa.