



PYTHON – AULA 2

LÓGICA DE PROGRAMAÇÃO APLICADA



AGENDA

Aula 1

Instalando e conhecendo o Python e as ferramentas do curso.
Revisitando lógica de programação em Python.

Aula 2

Revisitando lógica de programação em Python –
continuação.

Aula 3

Trabalhando com listas, tuplas e dicionários.
Leitura e escrita de JSON.

Aula 4

Leitura e escrita de arquivos.
Introdução a Numpy e Pandas.

AGENDA

Aula 5

Orientação a objeto no Python.

Aula 6

Trabalhando com os algoritmos.
Ordenação e Recursão. – Pesquisa em largura.

Aula 7

Algoritmo de Dijkstra.

Aula 8

Algoritmos Genéticos.



TRABALHANDO COM LAÇOS DE REPETIÇÃO






PROGRAMANDO EM PYTHON



CONHECENDO O LOOP
WHILE



Pense no seguinte problema:

Criar um script que valide o usuário “admin” e a senha “123”.

Esse script deve ser executado infinitas vezes até que o usuário acerte o login e a senha propostos. O que podemos fazer?

A primeira parte do problema é simples, basta criarmos duas variáveis para a digitação do usuário e um desvio condicional para a validação dos dados.

Vamos começar fazendo isso em um novo script chamado `loop_while.py`.



```
username = input("Por favor, insira seu nome de usuário")  
password = input("Por favor, insira sua senha")
```

```
if(username.upper() == "ADMIN" and password.upper() ==  
"123"):  
    print("Você está logado no sistema")  
else:  
    print("Dados de acesso incorretos")
```


- Estamos esquecendo, porém, da segunda parte do nosso problema: permitir que o usuário tente novamente infinitas vezes até que ele acerte.

Uma solução temporária, nada elegante e apenas parcialmente correta seria recorrermos a muitos desvios condicionais encadeados.

Vamos ver essa solução antes de descartarmos:

```

• • • username = input("Por favor, insira seu nome de usuário")
• • • password = input("Por favor, insira sua senha")

• + • if(username.upper() == "ADMIN" and password.upper() == "123"):
      print("Você está logado no sistema")
+ • else:
      username = input("Por favor, insira seu nome de usuário")
      password = input("Por favor, insira sua senha")

|
+
      if (username.upper() == "ADMIN" and password.upper() == "123"):
          print("Você está logado no sistema")
      else:
          username = input("Por favor, insira seu nome de usuário")
          password = input("Por favor, insira sua senha")

      if (username.upper() == "ADMIN" and password.upper() == "123"):
          print("Você está logado no sistema")
      else:
          username = input("Por favor, insira seu nome de usuário")
          password = input("Por favor, insira sua senha")

      if (username.upper() == "ADMIN" and password.upper() == "123"):
          print("Você está logado no sistema")
      else:
          print("Dados de acesso incorretos")

```



O uso de diversos desvios condicionais está permitindo que o usuário tente acessar o sistema por 4 vezes.

Porém, se quiséssemos ampliar esse número, o método se revelaria bastante ineficaz.



- Por mais que a solução inicial não seja a mais adequada, ela nos apontou para o caminho certo: **repetição**.

Os **Laços de Repetição ou Loops** são estruturas que nos permitem repetir a execução de determinados trechos do nosso código e essas repetições podem ser controladas por diferentes critérios.



O primeiro loop que estudaremos é o loop **while**.

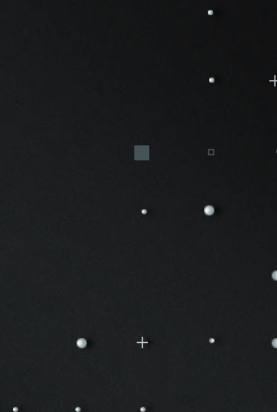
Trata-se de um loop baseado em *condições* e as instruções são executadas enquanto essas condições continuarem sendo **verdadeiras**.

As condições que um loop while suporta também são booleanas, assim como ocorria com os desvios condicionais.



|
+

```
while(condição):  
    //instruções que serão repetidas
```



- Para entendermos bem o funcionamento desse loop, vamos criar no nosso programa anterior uma variável booleana que iniciará com valor FALSO.

Essa variável só mudará de valor quando o usuário acertar o login.

Nosso loop, por sua vez, terá como condição de execução a variável booleana.



```
username = input("Por favor, insira seu nome de usuário")
password = input("Por favor, insira sua senha")

login_realizado = False

while(login_realizado is False):
    if(username.upper() == "ADMIN" and password.upper() == "123"):
        print("Você está logado no sistema")
        login_realizado = True
    else:
        print("Dados de acesso incorretos")
        username = input("Por favor, insira seu nome de usuário")
        password = input("Por favor, insira sua senha")
```

Que tal modificarmos esse script para que, ao final, o usuário seja informado sobre quantas tentativas ele precisou utilizar para acertar o login e a senha?



```
username = input("Por favor, insira seu nome de usuário")
password = input("Por favor, insira sua senha")
```

```
login_realizado = False
tentativas = 0
```

```
while(login_realizado is False):
```

```
    tentativas = tentativas + 1
```

```
    if(username.upper() == "ADMIN" and password.upper() == "123"):
```

```
        print("Você está logado no sistema")
```

```
        login_realizado = True
```

```
    else:
```

```
        print("Dados de acesso incorretos")
```

```
        username = input("Por favor, insira seu nome de usuário")
```

```
        password = input("Por favor, insira sua senha")
```

```
    print("Para acessar o sistema foram utilizadas {}  
tentativas!".format(tentativas))
```

O loop while é extremamente poderoso e útil, pois podemos criar condições para praticamente qualquer cenário em que necessitarmos de repetições.


Para casos onde não quisermos usar repetições, porém, mas intervalos, existe um loop muito mais adequado.



PROGRAMANDO EM PYTHON



CONHECENDO O LOOP
FOR



- O loop For é um laço de repetição baseado em contadores e intervalos.

Isso quer dizer que, ao contrário do while, não estabelecemos condições para que uma repetição ocorra, mas sim um intervalo de valores em que ela deve acontecer.

Ficou confuso? Vamos pensar em um exemplo lúdico:



O loop while pode ser comparado à tarefa de fazer um bolo: nós continuamos batendo *enquanto* a massa ainda não está homogênea. Não sabemos de antemão quantas batidas serão necessárias.

O loop for, por sua vez, pode ser comparado a uma corrida de fórmula 1: a repetição do trajeto da pista deve ser feita por uma determinada quantidade de vezes. Nós temos o conhecimento e o controle prévio de quantas voltas serão dadas.



```
|  
+  
  
for <variável_contadora> in <intervalo>:  
    //instruções a serem repetidas
```

Vamos começar com um exemplo simples: queremos exibir na tela do computador todos os números inteiros entre 0 e 999.

Evidentemente poderíamos utilizar 1000 linhas de código com um comando print, mas já conhecemos a existência dos loops.

Crie um script chamado `loop_for.py`.



```
for contador in range(0,1000, 1):  
    print("Nessa volta o contador possui o valor:  
{0}".format(contador))
```

A função *range* nos retorna um intervalo que vai de 0 até 999, de 1 em 1.

O primeiro argumento dessa função é o valor inicial (0).

O segundo argumento dessa função é o valor final (999, pois o 1000 não será incluído).

O terceiro argumento dessa função é o incremento (1, o que quer dizer que os valores no intervalo são espaçados de 1 em 1).

· Agora que conhecemos esse loop e a função *range*, vamos exercitar um pouco mais nossas habilidades.

· Crie um script chamado *tabuada_for.py* e crie um programa que receba do usuário um número inteiro e calcule e exiba a tabuada desse número.



```
numero = int(input("Digite o número do qual deseja obter a  
tabuada"))
```

```
for multiplicador in range(1,11, 1):  
    resultado = multiplicador * numero  
    print("{} x {} = {}".format(multiplicador, numero, resultado))
```


Os loops serão extremamente úteis e necessários para a continuidade do nosso estudo!

Tente realizar com o loop while os mesmos programas que realizou com o loop for para se apropriar das diferenças entre ambos.


Na próxima aula conheceremos algumas estruturas de dados e compreenderemos como elas podem ser usadas juntamente com os loops!



PROGRAMANDO EM PYTHON



TRABALHANDO COM
FUNÇÕES SEM RETURN



Para entendermos bem a importância e o uso dos métodos e funções em linguagem de programação, vamos primeiro analisar um código que não utiliza esses recursos (mas poderia se beneficiar muito deles).

Um simples programa de Calculadora, com um menu que possibilita ao usuário as opções DIGITAR 2 VALORES, SOMAR, SUBTRAIR, DIVIDIR, MULTIPLICAR ou SAIR DO PROGRAMA.

Vejamos

```

opcao = 0
while opcao != 6:
    print("PROGRAMA CALCULADORA")
    print("Escolha sua opção!")
    print("1 - Digitar valores")
    print("2 - Realizar soma")
    print("3 - Realizar subtração")
    print("4 - Realizar divisão")
    print("5 - Realizar multiplicação")
    print("6 - Sair")
    opcao = int(input("\nPor favor, insira sua opção:\n"))
    if opcao == 1:
        valor1 = float(input("Digite o 1º valor"))
        valor2 = float(input("Digite o 2º valor"))
        print("Os valores {} e {} foram armazenados\n\n".format(valor1, valor2))
    elif opcao == 2:
        print("\n\nRealizando a soma entre {} e {}".format(valor1, valor2))
        soma = valor1 + valor2
        print("O resultado foi {}!\n\n".format(soma))
    elif opcao == 3:
        print("\n\nRealizando a subtração entre {} e {}".format(valor1, valor2))
        subtracao = valor1 - valor2
        print("O resultado foi {}!\n\n".format(subtracao))
    elif opcao == 4:
        print("\n\nRealizando a divisão entre {} e {}".format(valor1, valor2))
        divisao = valor1 / valor2
        print("O resultado foi {}!\n\n".format(divisao))
    elif opcao == 5:
        print("\n\nRealizando a multiplicação entre {} e {}".format(valor1, valor2))
        multiplicacao = valor1 * valor2
        print("O resultado foi {}!\n\n".format(multiplicacao))
    elif opcao == 6:
        print("Saindo do sistema...")
    else:
        print("Opção inválida")

```



São tantas linhas de código que fica até difícil exibir em uma única página, não é?

E esse é um sintoma de um problema que buscaremos resolver.

Nesse momento todas as **funcionalidades** do nosso programa encontram-se dentro do mesmo bloco de código.

Ou melhor dizendo: todas as funcionalidades do nosso programa estão *soltas* dentro de um único script linear.

Isso pode representar sérios problemas, dentre eles a manutenção do código.

Imagine que você queira realizar uma alteração na *mensagem* que é exibida ao final de cada operação matemática.

Teria que realizar alterações nas linhas 19, 23, 27 e 31, correndo o **risco de esquecer uma delas** ou de realizar alterações diferentes.

Além disso, um programador que estivesse com pressa poderia acabar **alterando alguma linha que não tem nada a ver** com o problema que ele gostaria de resolver.

É para isso que existe o conceito de *modularização*.

A ideia por trás desse conceito é criar, dentro de um programa, *subprogramas* responsáveis por resolver problemas específicos.

Quando o *programa principal* precisar utilizar as funcionalidades, ele apenas *chama* o subprograma.

Esses subprogramas são chamados *de métodos ou funções*.

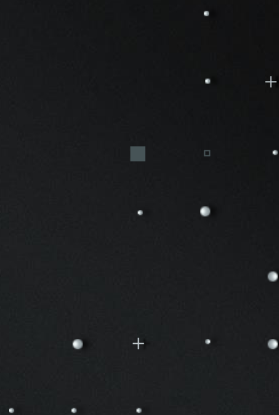
IDENTIFICANDO UMA FUNÇÃO EM POTENCIAL

```
1 opcao = 0
2 while opcao != 6:
3     print("PROGRAMA CALCULADORA")
4     print("Escolha sua opção!")
5     print("1 - Digitar valores")
6     print("2 - Realizar soma")
7     print("3 - Realizar subtração")
8     print("4 - Realizar divisão")
9     print("5 - Realizar multiplicação")
10    print("6 - Sair")
11    opcao = int(input("\nPor favor, insira sua opção:\n"))
12    if opcao == 1:
13        valor1 = float(input("Digite o 1º valor"))
14        valor2 = float(input("Digite o 2º valor"))
15        print("Os valores {} e {} foram armazenados\n\n".f
16    elif opcao == 2:
17        print("\n\nRealizando a soma entre {} e {}".format
18        soma = valor1 + valor2
19        print("O resultado foi {}!\n\n".format(soma))
20    elif opcao == 3:
21        print("\n\nRealizando a subtração entre {} e {}".f
```

- Observe o trecho de código ao lado, presente no projeto em que estamos trabalhando.
- O código das linhas 2 até 10 tem um objetivo específico: **exibir as opções de menu do nosso programa.**
- Podemos, portanto, criar uma **função** que cumprirá apenas esse mesmo objetivo!



```
| def nome_da_funcao(argumentos da função):  
+ //instruções que serão executadas quando a função for chamada
```





```
def exibe_menu():  
    print("PROGRAMA CALCULADORA")  
    print("Escolha sua opção!")  
    print("1 - Digitar valores")  
    print("2 - Realizar soma")  
    print("3 - Realizar subtração")  
    print("4 - Realizar divisão")  
    print("5 - Realizar multiplicação")  
    print("6 - Sair")
```

Se criarmos a função `exibe_menu()` dentro do nosso script e escrevermos dentro dela todo o conteúdo que desejarmos que ela execute, quando a função for chamada esse conteúdo será executado.

Agora que nossa função já está escrita, precisaremos resolver 2 problemas:

1 – Remover o código que exibia as mensagens do menu dentro da main, logo após o início do loop

2 – Fazer a chamada da função que exibe as mensagens.


```

opcao = 0
while opcao != 6:
    exibe_menu()
    opcao = int(input("\nPor favor, insira sua opção:\n"))
    if opcao == 1:
        valor1 = float(input("Digite o 1º valor"))
        valor2 = float(input("Digite o 2º valor"))
        print("Os valores {} e {} foram armazenados\n\n".format(valor1, valor2))
    elif opcao == 2:
        print("\n\nRealizando a soma entre {} e {}".format(valor1, valor2))
        soma = valor1 + valor2
        print("O resultado foi {}".format(soma))
    elif opcao == 3:
        print("\n\nRealizando a subtração entre {} e {}".format(valor1, valor2))
        subtracao = valor1 - valor2
        print("O resultado foi {}".format(subtracao))
    elif opcao == 4:
        print("\n\nRealizando a divisão entre {} e {}".format(valor1, valor2))
        divisao = valor1 / valor2
        print("O resultado foi {}".format(divisao))
    elif opcao == 5:
        print("\n\nRealizando a multiplicação entre {} e {}".format(valor1, valor2))
        multiplicacao = valor1 * valor2
        print("O resultado foi {}".format(multiplicacao))
    elif opcao == 6:
        print("Saindo do sistema...")
    else:
        print("Opção inválida")

```





```
def exibe_resultado():  
    print("O resultado foi ")
```

O esqueleto da função *exibe_resultado()* será parecido com esse acima: uma função que exibe uma mensagem.

Note, porém, que nossa função precisa fazer uso de uma informação que existe no fluxo principal do nosso programa.

A melhor forma de lidarmos com essa necessidade prévia de valores é usando argumentos.

É claramente mais vantajoso utilizarmos *argumentos* para indicar quais informações precisam ser fornecidas para que a função execute!

No nosso caso, criaremos como argumento o resultado da operação.



```
def exibe_resultado(resultado):  
    print("O resultado foi {}".format(resultado))
```

Ao indicarmos, entre parênteses o parâmetro chamado *resultado* estamos informando ao Python que para que essa função seja executada será obrigatório informar um valor.

Esse valor será armazenado em *resultado* e poderá ser utilizado dentro da função *exibe_resultado()*



```
elif opcao == 2:
    print("\n\nRealizando a soma entre {} e {}".format(valor1, valor2))
    soma = valor1 + valor2
    exibe_resultado(soma)
elif opcao == 3:
    print("\n\nRealizando a subtração entre {} e {}".format(valor1, valor2))
    subtracao = valor1 - valor2
    exibe_resultado(subtracao)
elif opcao == 4:
    print("\n\nRealizando a divisão entre {} e {}".format(valor1, valor2))
    divisao = valor1 / valor2
    exibe_resultado(divisao)
elif opcao == 5:
    print("\n\nRealizando a multiplicação entre {} e {}".format(valor1, valor2))
    multiplicacao = valor1 * valor2
    exibe_resultado(multiplicacao)
```



PROGRAMANDO EM PYTHON



TRABALHANDO COM
FUNÇÕES COM RETURN



Já aprendemos a modularizar nossos programas, seja usando argumentos ou não.

É chegada a hora de darmos mais um poder especial às nossas funções: o retorno de dados.



```
def somar(valor1, valor2):  
    soma = valor1 + valor2  
    exibe_resultado(soma)
```

Uma função que soma dois valores poderia receber os valores como argumentos e, depois disso, chamar a função `exibe_resultado()` que criamos anteriormente, passando a soma como argumento.

Porém esse uso deixa nossa função restrita à situações em que queremos imprimir o resultado na tela. E nas situações onde isso não é desejável?

Quando desejamos que uma função, ao ser encerrada, devolva um dado para aquele trecho do código em que foi chamada, utilizamos o comando *return*.

O comando *return* encerra uma função e faz com que ela retorne algum dado.



```
def somar(valor1, valor2):  
    soma = valor1 + valor2  
    return soma
```

Agora a função *somar()* retorna um valor: o conteúdo da variável *soma*.

Isso quer dizer que quando o programa chegar até a linha *return soma* a função será encerrada e o valor da variável *soma* será devolvido para o mesmo lugar em que ocorreu a chamada da função *somar*.



```
elif opcao == 2:
    print("\n\nRealizando a soma entre {} e {}".format(valor1,
| valor2))
+ soma = somar(valor1, valor2)
    exibe_resultado(soma)
```

Como nossa função *somar()* agora retorna um valor, no momento em que chamamos essa função podemos indicar uma variável para receber o resultado.



```
elif opcao == 2:  
    print("\n\nRealizando a soma entre {} e {}".format(valor1, valor2))  
    exibe_resultado(somar(valor1, valor2))
```

Outra forma de chamarmos a função *somar()* é dentro da chamada da função *exibe_resultado()*: dessa forma a função *somar* será executada, devolverá um valor e o valor devolvido que será passado para a função *exibe_resultado()*

A grande vantagem de criar uma função que retorne dados é desvincular a utilização dessa função de um cenário específico (como exibir o resultado na tela).

Tente fazer o mesmo criando funções para subtrair, dividir e multiplicar!

BIBLIOGRAFIA BÁSICA

BEAZLEY, David. *Python Essential Reference*, 2009.

BHARGAVA, ADITYA Y. *Entendendo Algoritmos. Um guia ilustrado para programadores e outros curiosos*. São Paulo: Ed. Novatec, 2017

CORMEN, THOMAS H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2002

COSTA, Sérgio Souza. *Recursividade*. Professor Adjunto da Universidade Federal do Maranhão.

DOWNEY, ALLEN B. *Pense em Python. Pense como um cientista da computação*. São Paulo: Ed. Novatec, 2016

GRANATYR, Jones; PACHOLOK, Edson. *IA Expert Academy*. Disponível em: <https://iaexpert.academy/>

KOPEC, DAVID. *Problemas clássicos de ciência da computação com Python*. São Paulo: Ed. Novatec, 2019

LINDEN, Ricardo. *Algoritmos Genéticos*. 3 edição. Rio de Janeiro: Editora Moderna, 2012

MCKINNEY, WILLIAM WESLEY. *Python para análise de dados. Tratamento de dados com Pandas, Numpy e Ipython*. São Paulo: Ed. Novatec, 2018

BIBLIOGRAFIA BÁSICA

- TENEMBAUM, Aaron M. **Estrutura de Dados Usando C**. Sao Paulo: Makron Books do Brasil, 1995.
- VELLOSO, Paulo. **Estruturas de Dados**. Rio de Janeiro: Ed. Campus, 1991.
- VILLAS, Marcos V & Outros. **Estruturas de Dados: Conceitos e Tecnicas de implementacao**. RJ: Ed. Campus, 1993.
- PREISS, Bruno P. **Estrutura de dados e algoritmos: Padrões de projetos orientados a objetos com Java**. Rio de Janeiro: Editora Campus, 2001.
- PUGA, Sandra; RISSETTI, Gerson. **Lógica de programação e estruturas de dados**. 2016.
- SILVA, Osmar Quirino. **Estrutura de Dados e Algoritmos Usando C. Fundamentos e Aplicações**. Rio de Janeiro: Editora Ciência Moderna, 2007.
- ZIVIANI, N. **Projeto de algoritmos com implementações em pascal e C**. São Paulo: Editora Thomsom, 2002.

OBRIGADO



FIAP

Copyright © 2023 | Professor Dr. Emerson R. Abraham

Todos os direitos reservados. A reprodução ou divulgação total ou parcial deste documento é expressamente proibida sem o consentimento formal, por escrito, do professor/autor.

