



DEFINIÇÃO

Identificação de estruturas de controle, de decisão e de repetição na linguagem Python, bem como de conceitos e da implementação de subprogramas e bibliotecas e das formas de tratamento de exceções e eventos.

PROPÓSITO

Reconhecer, na linguagem Python, as estruturas de decisão e de repetição, a utilização de subprogramas e de bibliotecas e as formas de tratamento de exceção e eventos.

PREPARAÇÃO

Antes de iniciar o conteúdo deste tema, é necessário que tenha o interpretador Python na versão 3.7.7 e o ambiente de desenvolvimento PyCharm ou outro ambiente que suporte o

desenvolvimento na linguagem Python.

É necessário conhecer tipos de variáveis em Python, como também realizar a entrada e saída de dados em Python.

OBJETIVOS

MÓDULO 1

Descrever as estruturas de decisão e repetição em Python

MÓDULO 2

Definir os principais conceitos de subprogramas e a sua utilização em Python

MÓDULO 3

Identificar o uso correto de recursos de bibliotecas em Python

MÓDULO 4

Analisar as formas de tratamento de exceções e eventos em Python



📷 Fonte: BEST-BACKGROUNDS | Shutterstock

INTRODUÇÃO

Programar significa, como em qualquer disciplina, **aprender ferramentas que permitam desenvolver melhor a sua atividade**. Ao aprender os conceitos básicos de programação, o estudante desenvolve habilidades iniciais para escrever seus primeiros programas. Porém, é difícil imaginar que aplicações profissionais sejam feitas totalmente baseadas apenas nesses conceitos básicos.

Ao pensarmos em **aplicações mais complexas**, é essencial considerar a necessidade de **ganhar tempo**, com o computador executando as tarefas repetitivas, e as demandas de manutenção e tratamento de erros. Para avançar no aprendizado da programação, você conhecerá novas ferramentas, entre elas as **estruturas de controle**, como decisão e repetição, além dos **subprogramas e bibliotecas**, bem como **as formas de tratar exceções e eventos**.

Vamos começar nossa jornada acessando os códigos-fontes originais propostos para o aprendizado de Python estruturado. **Baixe o arquivo aqui**, descompactando-o em seu dispositivo. Assim, você poderá utilizar os códigos como material de apoio ao longo do tema!

MÓDULO 1

🕒 Descrever as estruturas de decisão e repetição em Python



📷 Fonte: dTosh | Shutterstock

As estruturas de controle permitem selecionar quais partes do código serão executadas – chamadas de **estruturas de decisão** – e repetir blocos de instruções com base em algum critério, como uma variável de controle ou a validade de alguma condição – chamadas de **estruturas de repetição**. Neste módulo, vamos conhecer as estruturas de decisão e de repetição em Python.

TRATAMENTO DAS CONDIÇÕES

As estruturas de decisão e de repetição possuem sintaxes bastante semelhantes em C e em Python. Mesmo com essa grande semelhança, existe uma diferença crítica no tratamento das condições. **Diferentemente da linguagem C, Python oferece o tipo bool**. Por isso, cabe ressaltar a diferença de comportamento das duas linguagens nesse tratamento.

Python	C
Existe o tipo bool	Não existe o tipo bool
True	Qualquer valor diferente de 0 (zero)
False	0 (zero) ou vazio

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Tabela 1 - Tratamento das condições - Fonte: O autor (2020)

ATENÇÃO

Observe que o fato de haver o tipo bool em Python permite que as condições sejam tratadas como **verdadeiras** ou **falsas**, o que não é exatamente igual em C.

AS ESTRUTURAS DE DECISÃO IF, IF-ELSE E ELIF

Em Python, é possível utilizar as estruturas de decisão **if** e **if-else** da mesma forma que em C. A diferença principal é o modo de delimitar os blocos de instruções relativos a cada parte da estrutura. Observe a Tabela 2 e a Tabela 3:

Python	C
if <condição>:	if <condição>{
Instruções com 4 espaços de indentação	A indentação não é exigida
Instrução fora do if	}

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Tabela 2 - Estruturas de decisão simples - Fonte: O autor (2020)

Python	C
if <condição>:	if <condição>{
Instruções com 4 espaços de indentação (caso a condição seja verdadeira)	Bloco 1 (caso a condição seja verdadeira). A indentação não é exigida

else:	} else {
Instruções com 4 espaços de indentação (caso a condição seja falsa)	Bloco 2 (caso a condição seja falsa). A indentação não é exigida
Instrução fora do if	}

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Tabela 3 - Estruturas de decisão compostas - Fonte: O autor (2020)

Python também oferece a estrutura **elif**, que permite o teste de duas condições de forma sequencial. Essa estrutura não existe em C, sendo necessário o encadeamento de estruturas **if-else**. Em geral, o formato da estrutura **elif** é:

- 1 if <condição 1>:
- 2 Bloco de código que será executado caso condição seja True
- 3 elif <condição 2>:
- 4 Bloco de código que será executado caso condição 1 seja False e condição 2 seja True
- 5 else:
- 6 Bloco de código que será executado caso condição 1 seja False e condição 2 seja False
- 7 Instrução fora do if

Veja uma implementação possível com a estrutura **elif** na Figura 1:

```

1 idade = eval(input('Informe a idade da criança: '))
2 if idade < 5:
3     print('A criança deve ser vacinada contra a gripe.')
4     print('Procure o posto de saúde mais próximo.')
5 elif idade == 5:
6     print('A vacina estará disponível em breve.')
7     print('Aguarde as próximas informações.')
8 else:
9     print('A vacinação só ocorrerá daqui a 3 meses.')
10    print('Informe-se novamente neste prazo.')
11    print('Cuide da saúde sempre. Até a próxima.')
```

Figura 1 - A estrutura elif - Fonte: O autor (2020)

Perceba que a indentação precisa ser ajustada, uma vez que o último else é relativo ao elif. Por isso, eles precisam estar alinhados.

ESTRUTURA DE REPETIÇÃO FOR

A estrutura de repetição **for** tem funcionamento muito semelhante nas linguagens C e Python. Porém, a sintaxe é diferente nas duas linguagens. Além disso, em Python existe maior flexibilidade, já que a repetição pode ser controlada por uma variável não numérica.

Antes de detalhar o **for**, vamos conhecer uma função de Python que gera uma **lista de valores numéricos**. Essa lista ajudará a verificar a repetição e deixará mais claro o entendimento do laço.

AS LISTAS DO TIPO RANGE()

Ao chamar o método `range()`, Python cria uma sequência de números inteiros, de maneira simples à mais complexa. Veja a seguir:

SIMPLES

NÃO INICIADAS EM 0

INDICANDO INÍCIO, FIM E PASSO

SIMPLES

Ela pode ser chamada de **maneira simples**, apenas com um argumento. Nesse caso, a sequência começará em 0 e será incrementada de uma unidade até o limite do parâmetro passado (**exclusive**).

Por exemplo: `range(3)` cria a sequência (0, 1, 2).

NÃO INICIADAS EM 0

Para que a sequência não comece em 0, podemos informar o início e o fim como parâmetros, lembrando que o parâmetro fim não entra na lista (exclusive o fim). O padrão é incrementar cada termo em uma unidade. Ou seja, a chamada **range(2, 7)** cria a sequência (2, 3, 4, 5, 6).

INDICANDO INÍCIO, FIM E PASSO

Também é possível criar sequências mais **complexas**, indicando os parâmetros de início, fim e passo, nessa ordem. **O passo é o valor que será incrementado de um termo para o próximo.**

Por exemplo, **range(2, 9, 3)** cria a sequência (2, 5, 8).

A SINTAXE DA ESTRUTURA FOR

A estrutura **for** tem a seguinte sintaxe em Python:

- 1 for <variável> in <sequência>:
- 2 Bloco que será repetido para todos os itens da sequência
- 3 Instrução fora do for

Cabe ressaltar a diferença de sintaxe entre as linguagens C e Python. Veja a Tabela 4:

Python	C
for <variável> in <sequência>:	for (inicialização; condição; incremento ou decremento){
Instruções com 4 espaços de indentação	Bloco de instruções a ser repetido. A indentação não é exigida

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Tabela 4 - A estrutura for - Fonte: O autor (2020)

Vamos analisar um exemplo simples em Python: imprimir todos os elementos de uma sequência criada com a chamada **range()**. Veja uma possível implementação desse exemplo na Figura 2:

```
1 for item in range(2, 9, 3):  
2 print(item)
```

Figura 2 - Um exemplo do for em Python - Fonte: O autor (2020)

A **linha 1** mostra a criação do laço, com a variável **item** percorrendo a sequência (2, 5, 8), criada pela chamada **range(2, 9, 3)**;

A **linha 2** indica a instrução que será executada para cada repetição deste laço. O laço **for** executa a instrução da **linha 2** três vezes, uma para cada elemento da sequência (2, 5, 8); o resultado está exemplificado na Tabela 5.

	sequência	2	5	8
Iteração 1 do laço	item =	2		
Iteração 2 do laço	item =		5	
Iteração 3 do laço	item =			8

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Tabela 5 - Um exemplo do for em Python - Fonte: O autor (2020))

O LAÇO FOR COM UMA STRING

Python também permite que a repetição aconteça ao longo de uma **string**. Para isso, basta lembrar que **a string é uma sequência de caracteres individuais**. Suponha que você queira soletrar o nome informado pelo usuário. Uma possível implementação está na Figura 3:

```
1 nome = input("Entre com seu nome: ")
2 for letra in nome:
3     print(letra)
```

Figura 3 - Uso do for com uma string - Fonte: O autor (2020)

A **linha 1** faz com que a palavra inserida pelo usuário seja armazenada na variável *nome*;

A **linha 2** mostra a criação do laço, com a variável **letra** percorrendo a sequência de caracteres armazenada na variável **nome**;

A **linha 3** indica a instrução que será executada para cada repetição desse laço. O laço **for** executa a instrução da **linha 3** tantas vezes quantos forem os elementos da sequência que está na variável **nome**.

Veja um exemplo de execução na Figura 4:

```
1 Entre com o seu nome: Laura
2 L
3 a
4 u
5 r
6 a
```

Figura 4 - Execução do for com uma string - Fonte: O autor (2020)

USO DO LAÇO FOR COM QUALQUER SEQUÊNCIA

Até agora, estudamos o uso do laço **for** com iterações sobre strings e sobre sequências numéricas, mas Python permite ainda mais que isso!

PODEMOS UTILIZAR O LAÇO FOR COM ITERAÇÕES SOBRE QUALQUER SEQUÊNCIA, NÃO SOMENTE AS NUMÉRICAS E AS STRINGS.

Observe o exemplo da Figura 5:

```
1 nomes = ['Laura', 'Lis', 'Guilherme', 'Enzo', 'Arthur']  
2 for nome in nomes:  
3     print(nome)
```

Figura 5 - Uso do for com qualquer sequência - Fonte: O autor (2020)

Veja o resultado da execução na Figura 6:

```
1 Laura  
2 Lis  
3 Guilherme  
4 Enzo  
5 Arthur
```

Figura 6 - Exemplo de execução do for com qualquer sequência - Fonte: O autor (2020)

ESTRUTURA DE REPETIÇÃO WHILE

A estrutura de repetição **while** tem funcionamento e sintaxe muito semelhantes nas linguagens C e Python. Observe a comparação entre as duas linguagens na Tabela 6:

Python	C
<code>while <condição>:</code>	<code>while <condição>{</code>
Instruções com 4 espaços de indentação	Bloco de instruções a ser repetido. A indentação não é exigida
Instrução fora do while	<code>}</code>

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Tabela 6 - Comparação do while em Python e em C - Fonte: O autor (2020)

Como exemplo inicial do uso do laço **while**, vamos analisar um programa em que o usuário precisa digitar a palavra “sair” para que o laço **while** seja encerrado.

Uma possível implementação desse exemplo em Python está na Figura 7:

```
1 palavra = input('Entre com uma palavra: ')
2 while palavra != 'sair':
3     palavra = input('Digite sair para encerrar o laço: ')
4     print('Você digitou sair e agora está fora do laço')
```

Figura 7 - Um exemplo de implementação do while em Python - Fonte: O autor (2020)

A **linha 1** representa a solicitação ao usuário para que ele insira uma palavra, que será armazenada na variável **palavra**;

A **linha 2** cria o laço **while**, que depende da condição <valor da variável **palavra** ser diferente de 'sair'>;

A **linha 3** será repetida enquanto a condição for verdadeira, ou seja, enquanto o valor da variável **palavra** for diferente de 'sair'. Quando esses valores forem iguais, a condição do laço **while** será falsa e o laço será encerrado;

A **linha 4** representa a impressão da mensagem fora do laço **while**.

Veja uma execução desse programa na Figura 8:

```
1 Entre com uma palavra: teste
2 Digite sair para encerrar o laço: Oi?
3 Digite sair para encerrar o laço: Estou tentando...
4 Digite sair para encerrar o laço: Aff...
5 Digite sair para encerrar o laço: Blz, entendi...
6 Digite sair para encerrar o laço: Sair
7 Digite sair para encerrar o laço: Ah! o 'S' foi maiúsculo
8 Digite sair para encerrar o laço: sair
9 Você digitou sair e agora está fora do laço
```

Figura 8 - Uma execução do while em Python - Fonte: O autor (2020)

Observe agora outra execução do mesmo programa na Figura 9:

1 Entre com uma palavra: *sair*

2 Você digitou sair e agora está fora do laço

Figura 9 - Outra execução do while em Python - Fonte: O autor (2020)

Perceba que ao digitar 'sair' logo na primeira solicitação, a **linha 3** do nosso programa não é executada nenhuma vez. Ou seja, o programa nem chega a entrar no laço **while**.

Em C, existe outra estrutura muito semelhante ao **while**, chamada **do-while**. A diferença básica entre elas é o momento em que a condição é testada, como vemos a seguir:

No laço **while**, a condição é testada antes da iteração.

O laço **while** testa e executa caso a condição seja verdadeira.



No laço **do-while**, a condição é testada após a iteração.

O laço **do-while** executa e testa.

Infelizmente, a estrutura **do-while não existe em Python**. Isso não chega a ser um grande problema, porque podemos adaptar nosso programa e controlar as repetições com o laço **while**.

O LAÇO WHILE INFINITO

Laços infinitos são úteis quando queremos executar um bloco de instruções indefinidamente.

O laço **while** infinito tem o seguinte formato:

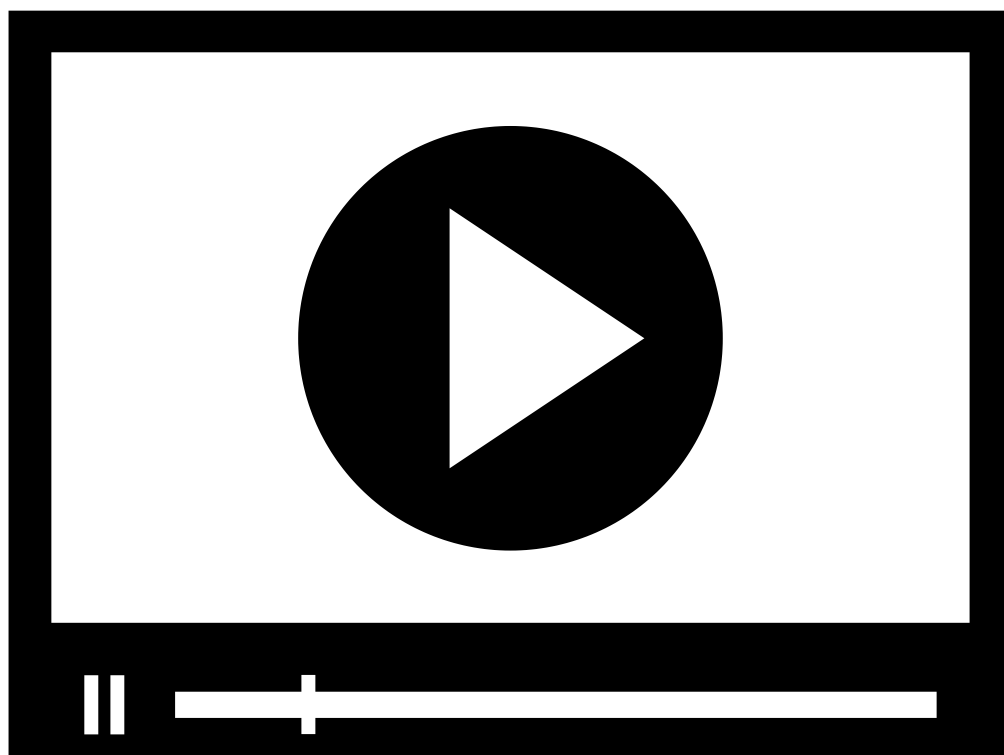
1 while True:

2 Bloco que será repetido indefinidamente

★ EXEMPLO

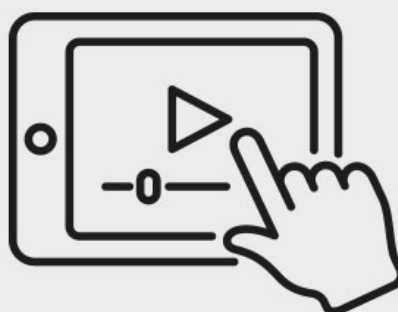
Suponha que você deseje criar uma aplicação que permaneça por meses ou anos sendo executada, registrando a temperatura ou a umidade de um ambiente. Logicamente, estamos supondo que você tenha essa informação disponível a partir da leitura de algum sensor.

Deve-se tomar cuidado e ter certeza de que seu uso é realmente necessário para evitar problemas de consumo excessivo de memória.



No vídeo a seguir, o professor nos apresenta exemplos práticos do uso das estruturas de decisão e repetição. Vamos assistir!

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



AS INSTRUÇÕES AUXILIARES BREAK, CONTINUE E PASS

A INSTRUÇÃO BREAK

A instrução **break** funciona da mesma maneira em C e em Python. **Ela interrompe as repetições dos laços for e while.** Quando a execução do programa chega a uma instrução **break**, a repetição é encerrada e o fluxo do programa segue a partir da primeira instrução seguinte ao laço.

Para exemplificar o uso da instrução **break**, vamos voltar ao primeiro exemplo do laço **while**, utilizando o laço infinito. O laço será encerrado quando o usuário inserir a palavra 'sair'. Veja a Figura 10:

```
1 while True:
2     palavra = input('Entre com uma palavra: ')
3     if palavra == 'sair':
4         break
5     print('Você digitou sair e agora está fora do laço')
```

Figura 10 - Um exemplo de uso do break em Python - Fonte: O autor (2020)

Caso haja vários laços aninhados, o **break** será relativo ao laço em que estiver inserido. Veja a Figura 11:

```
1 while True:
2     print('Você está no primeiro laço.')
3     opcao1 = input('Deseja sair dele? Digite SIM para isso. ')
4     if opcao1 == 'SIM':
5         break # este break é do primeiro laço
6     else:
7         while True:
8             print('Você está no segundo laço.')
9             opcao2 = input('Deseja sair dele? Digite SIM para isso. ')
10            if opcao2 == 'SIM':
11                break # este break é do segundo laço
12            print('Você saiu do segundo laço.')
13            print('Você saiu do primeiro laço')
14
```

Figura 11 - Uso do break em laços aninhados - Fonte: O autor (2020)

A INSTRUÇÃO CONTINUE

A instrução **continue** também funciona da mesma maneira em C e em Python. Ela atua sobre as repetições dos laços **for** e **while**, como a instrução **break**, mas não interrompe todas as repetições do laço. **A instrução continue interrompe apenas a iteração corrente, fazendo com que o laço passe para a próxima iteração.**

O exemplo a seguir imprime todos os números inteiros de 1 até 10, pulando apenas o 5. Veja sua implementação na Figura 12:

```
1 for num in range(1, 11):  
2     if num == 5:  
3         continue  
4     else:  
5         print(num)  
6     print('Laço encerrado')  
7
```

Figura 12 - Exemplo de uso do continue em Python - Fonte: O autor (2020)

Para ressaltar a diferença entre as instruções **break** e **continue**, vamos alterar a **linha 3** do nosso programa, trocando a instrução **continue** pela instrução **break**. Veja a nova execução na Figura 13:

```
1  
2  
3  
4  
Laço encerrado
```

Figura 13 - Troca do continue pelo break - Fonte: O autor (2020)

A INSTRUÇÃO PASS

A instrução **pass** atua sobre a estrutura **if**, permitindo que ela seja escrita sem outras instruções a serem executadas caso a condição seja verdadeira. Assim, podemos concentrar

as instruções no caso em que a condição seja falsa. Suponha que queiramos imprimir somente os números ímpares entre 1 e 10. Uma implementação possível está na Figura 14:

```
1 for num in range(1, 11):
2     if num % 2 == 0:
3         pass
4     else:
5         print(num)
6     print('Laço encerrado')
```

Figura 14 - Exemplo de uso do pass em Python - Fonte: O autor (2020)

Veja a execução desse programa na Figura 15:

```
1
3
5
7
9
Laço encerrado
```

Figura 15 - Exemplo de execução com uso do pass em Python - Fonte: O autor (2020)

Claramente, seria possível reescrever a condição do **if-else** para que pudéssemos transformá-lo em um **if** simples, sem **else**. Porém, o objetivo aqui é mostrar o uso da instrução **pass**.

Agora que já vimos os principais conceitos relativos às estruturas de decisão e de repetição, vamos testar seus conhecimentos.

VERIFICANDO O APRENDIZADO

1. CONSIDERE O SEGUINTE TRECHO DE UM PROGRAMA ESCRITO EM PYTHON:

```
1 S = 0
2 FOR I IN RANGE(5):
```

3 S += 3*I

4 PRINT(S)

ASSINALE A OPÇÃO QUE APRESENTA CORRETAMENTE O QUE SERÁ IMPRESSO NA TELA.

A) 0 3 9 18 30

B) 0 3 6 9 12

C) 30

D) 45

2. CONSIDERE O SEGUINTE TRECHO DE UM PROGRAMA ESCRITO EM PYTHON:

1 S = 0

2 A = 1

3 WHILE S < 5:

4 S = 3*A

5 A += 1

6 PRINT(S)

ASSINALE A OPÇÃO QUE APRESENTA CORRETAMENTE O QUE SERÁ IMPRESSO NA TELA.

A) 9

B) 3 6

C) 3 3

D) 3 6 9 12

GABARITO

1. Considere o seguinte trecho de um programa escrito em Python:

```
1 s = 0
2 for i in range(5):
3 s += 3*i
4 print(s)
```

Assinale a opção que apresenta corretamente o que será impresso na tela.

A alternativa "C " está correta.

O laço for vai ser repetido 5 vezes, já que **range(5)** retorna a sequência (0, 1, 2, 3, 4). Vale observar que a instrução **print(s)** está fora do laço **for**, o que a leva a ser executada apenas uma vez quando o laço se encerrar. Isso elimina as opções A e B. A variável **s** começa com valor zero e é acrescida, a cada iteração, do valor **3*i**, sendo que **i** pertence à sequência (0, 1, 2, 3, 4). Ou seja, **s recebe os acréscimos: 0 + 3 + 6 + 9 + 12. Assim, ela termina o laço com o valor 30, que será impresso pela instrução print(s).**

2. Considere o seguinte trecho de um programa escrito em Python:

```
1 s = 0
2 a = 1
3 while s < 5:
4 s = 3*a
5 a += 1
6 print(s)
```

Assinale a opção que apresenta corretamente o que será impresso na tela.

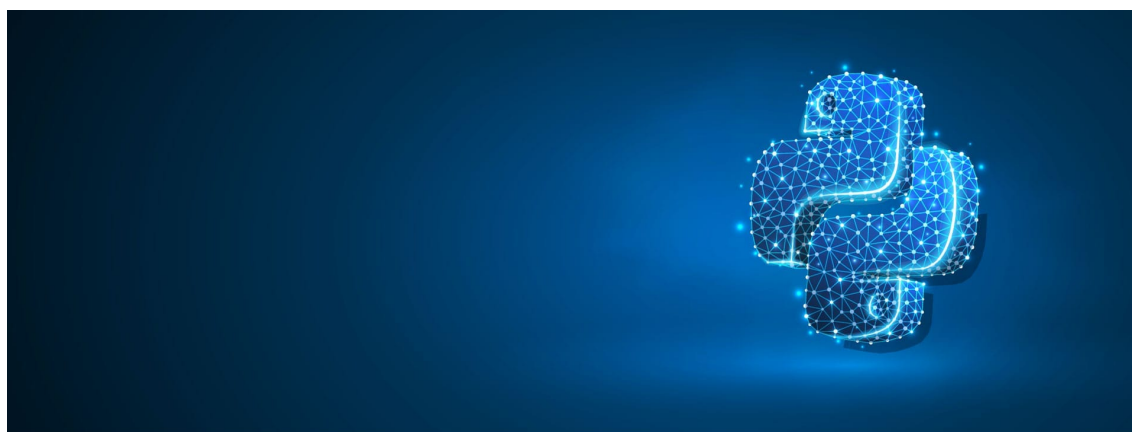
A alternativa "B " está correta.

Ao ser testada pela primeira vez, a condição do **while** é verdadeira, já que **s** vale zero. Assim, a variável **s** recebe o valor 3 (3×1) e a variável **a** é acrescida de uma unidade, ficando com o valor 2. Em seguida, é impresso o valor de **s** (3). A condição do **while** é, então, testada novamente, sendo mais uma vez verdadeira, porque **s** tem o valor 3 (menor que 5). Nessa iteração, a variável **s** recebe o valor 6 (3×2) e a variável **a** é acrescida de uma unidade, ficando com o valor 3. Em seguida, é impresso o valor de **s** (6). A condição do **while** é, então, testada

novamente e é falsa, já que **s** tem o valor 6, maior que 5. Com isso, o laço **while** é encerrado e nada mais é impresso. Logo, foram impressos os valores 3 e 6.

MÓDULO 2

- 🕒 Definir os principais conceitos de subprogramas e a sua utilização em Python



📷 Fonte: dTosh | Shutterstock

Os subprogramas são elementos fundamentais dos programas e por isso são importantes no estudo de linguagens de programação. Neste módulo, abordaremos os conceitos de subprogramas, como características gerais, passagem de parâmetros e recursividade, além da utilização de subprogramas em Python.

CARACTERÍSTICAS GERAIS DOS SUBPROGRAMAS

Todos os subprogramas estudados neste módulo, com base em Sebesta (2018), têm as seguintes características:

Cada subprograma tem um único ponto de entrada.

A unidade de programa chamadora é suspensa durante a execução do subprograma chamado, o que significa que existe apenas um subprograma em execução em determinado momento.

Quando a execução do subprograma termina, o controle sempre retorna para o chamador.

DEFINIÇÕES BÁSICAS

As definições básicas, conforme Sebesta (2018), estabelecem que:

1

Um **subprograma é definido** quando o desenvolvedor descreve a interface e as ações da abstração desse subprograma.

2

O **subprograma foi chamado** quando uma instrução traz um pedido explícito para sua execução.

3

O **subprograma está ativo** após o início de sua execução, a partir da sua chamada e enquanto ele não foi concluído.

O **cabeçalho** do subprograma é a primeira parte da definição, em que podem ser especificados o nome, os parâmetros e o tipo de retorno do subprograma.

Em C, o cabeçalho dos subprogramas – sendo chamados de funções – traz, em ordem: **o tipo de retorno, o nome e a lista de parâmetros**, como a seguir:

FLOAT CALCULAIMC (INT PESO, FLOAT ALTURA)

Em Python, as funções definidas pelo desenvolvedor devem ser precedidas pela palavra reservada **def**. Não são especificados o tipo de retorno nem os tipos dos parâmetros, como no exemplo a seguir:

DEF CALCULAIMC (PESO, ALTURA)

Em Python, as sentenças de função **def** são executáveis. Isso implica que a função só pode ser chamada após a execução da sentença **def**. Veja o exemplo na Figura 16:

```
1 escolha = input("Escolha uma opção de função: 1 ou 2")
```

```
2 if escolha == 1:
```

```
3 def func1(x):  
4     return x + 1  
5 else:  
6 def func2(x):  
7     return x + 2  
8  
9 s = func1(10)  
10 print(s)
```

Figura 16 - Funções em Python - Fonte: O autor (2020)

A função **func1()** só pode ser chamada caso a variável **escolha** seja igual a 1. Ou seja, o usuário deverá inserir 1 quando solicitado (**na linha 1**), para que a **linha 9** possa ser executada sem que seja gerado um erro.

PARÂMETROS

Usualmente, um subprograma executa cálculos e operações a partir de dados que ele deve processar. Existem duas maneiras de o subprograma obter esses dados: **acessando variáveis não locais, mas visíveis para o subprograma, ou pela passagem de parâmetros.**

Quando o subprograma recebe os parâmetros adequados, ele pode ser executado com quaisquer valores recebidos. Porém, quando ele manipula variáveis não locais, uma forma de evitar alterações indevidas nessas variáveis é **fazendo cópias locais delas**. De acordo com Sebesta (2018), o acesso sistemático a variáveis não locais pode diminuir a confiabilidade do programa.

São denominados **parâmetros formais** aqueles do cabeçalho do subprograma.

Quando o subprograma é chamado, é necessário escrever o nome do subprograma e a lista de parâmetros a serem vinculados aos parâmetros formais dele, que são denominados **parâmetros reais** ou **argumentos**.

No exemplo da Figura 16, existe o cabeçalho da função **func1** na **linha 3**, com o parâmetro formal **x**. Na **linha 9**, a função **func1** é chamada com o parâmetro real **10**.

Em Python, é possível estabelecer **valores padrão** para os parâmetros formais. O valor padrão é usado quando a chamada da função ocorre sem nenhum parâmetro real. Veja o exemplo de

definição e chamada da função **taxímetro** na Figura 17:

```
1 def taximetro(distancia, multiplicador=1):  
2     largada = 3  
3     km_rodado = 2  
4     valor = (largada + distancia * km_rodado) * multiplicador  
5     return valor  
6  
7  
8 pagamento = taximetro(3.5)  
9 print(pagamento)
```

Figura 17 - Exemplo de função com valor padrão - Fonte: o autor (2020)

Observe que mesmo com a definição da **linha 1** de dois parâmetros formais, a chamada da função na **linha 8** ocorre apenas com um parâmetro real.

A palavra reservada **return** indica que a função retorna algum valor. Isso implica que o valor retornado seja armazenado em uma variável do programa chamador (como ocorre na **linha 8**), ou utilizado como parâmetro para outra função.

ATENÇÃO

Retornar um valor é diferente de imprimir na tela. Ao utilizar a função **print()**, ocorre apenas a impressão de algo na tela, o que não significa que tenha havido retorno de qualquer função definida pelo usuário.

PROCEDIMENTOS E FUNÇÕES

Os subprogramas podem ser, distintamente, **procedimentos** e **funções**. De acordo com Sebesta (2018):

Procedimentos

São aqueles que não retornam valores.



Funções

São aquelas que retornam valores.

Na maioria das linguagens que não explicita a diferença entre eles, as funções podem ser definidas sem retornar qualquer valor, tendo comportamento de procedimento. Esse é o caso de Python. Veja o exemplo da Figura 18:

```
1 def func1(x):  
2 x = 10  
3 print(f'Função func1 - x = {x}')
```



```
4  
5  
6 def func2(x):  
7 x = 20  
8 print(f'Função func2 - x = {x}')
```



```
9  
10  
11 x = 0  
12 func1(x)  
13 func2(x)  
14 print(f'Programa principal - x = {x}')
```

Figura 18 - Procedimentos e funções - Fonte: O autor (2020)

As funções **func1(x)** e **func2(x)** não possuem qualquer retorno. Ou seja, são funções com comportamento de procedimentos.

AMBIENTES DE REFERENCIAMENTO LOCAL

VARIÁVEIS LOCAIS

Quando um subprograma define suas próprias variáveis, estabelece ambientes de referenciamento local. Essas variáveis são chamadas de **variáveis locais**, com seu escopo usualmente sendo o corpo do subprograma. As variáveis locais podem ser:

DINÂMICAS DA PILHA ESTÁTICAS

DINÂMICAS DA PILHA

São vinculadas ao armazenamento no início da execução do subprograma e desvinculadas quando essa execução termina. As variáveis locais dinâmicas da pilha têm diversas vantagens, e a principal delas é a flexibilidade. Suas principais desvantagens são o custo do tempo – para alocar, inicializar (quando necessário) e liberar tais variáveis para cada chamada ao subprograma – e o fato de que os acessos a essas variáveis locais devem ser indiretos, enquanto os acessos às variáveis estáticas podem ser diretos.

ESTÁTICAS

São vinculadas a células de memória antes do início da execução de um programa e permanecem vinculadas a essas mesmas células até que a execução do programa termine. Elas são um pouco mais eficientes que as variáveis locais dinâmicas da pilha, já que não é necessário tempo para alocar ou liberar essas variáveis. Sua maior desvantagem é a incapacidade de suportar recursão, como vai ser explicado adiante.

ATENÇÃO

Nas linguagens C e C++, as variáveis locais são dinâmicas da pilha, a menos que sejam especificamente declaradas como **static**. Todas as variáveis locais em Python são dinâmicas

da pilha. As variáveis globais são declaradas em definições de método, e qualquer variável declarada global em um método precisa ser definida fora dele. Caso haja uma atribuição à variável local com mesmo nome de uma variável global, esta é implicitamente declarada como local.

Voltando ao exemplo da Figura 18, vamos detalhar as funções **func1(x)** e **func2(x)**:

As **linhas 1, 2 e 3** definem a função **func1(x)**, que recebe o parâmetro **x**, mas tem uma variável local de nome **x**, cujo valor atribuído é 10;

Analogamente, a função **func2(x)** – definida nas **linhas 6, 7 e 8** – que recebe o parâmetro **x** e tem uma variável de mesmo nome, mas com valor atribuído 20;

O programa principal tem uma variável global de mesmo nome **x**, cujo valor atribuído é 0, na **linha 11**;

Veja que as chamadas às funções **func1(x)** e **func2(x)** ocorrem nas **linhas 12 e 13**, quando a variável **x** global já recebeu o valor 0. Porém, ao serem executadas, cada uma dessas funções tem a sua própria variável local, a quem todas as referências internas são feitas.

Confira a execução desse exemplo na Figura 19:

Função **func1** - **x** = 10

Função **func2** - **x** = 20

Programa principal - **x** = 0

Figura 19 - Execução de subprograma com variáveis locais - Fonte: O autor (2020)

Mesmo com a variável global tendo valor nulo, cada variável local das funções **func1(x)** e **func2(x)** tem seu próprio valor, e não ocorrem alterações na variável global mesmo com as atribuições das **linhas 2 e 7**.

Para alterar a variável global **x**, seria necessário explicitar dentro de cada função que o nome **x** é referente a ela. Isso pode ser feito com a palavra reservada **global**. Além de explicitar a referência à variável global, as funções **func1(x)** e **func2(x)** não recebem mais os parâmetros de mesmo nome, já que fazem referência à variável global. Veja como ficaria o nosso exemplo com essa pequena alteração na Figura 20:

```

1 def func1():
2     global x
3     x = 10
4     print(f'Função func1 - x = {x}')
5
6
7 def func2():
8     global x
9     x = 20
10    print(f'Função func2 - x = {x}')
11
12
13 x = 0
14 func1()
15 func2()
12 print(f'Programa principal - x = {x}')

```

Figura 20 - Exemplo de subprogramas com referência à variável global - Fonte: O autor (2020)

Observe agora a execução desse exemplo alterado na Figura 21:

Função func1 - x = 10

Função func2 - x = 20

Programa principal - x = 20

Figura 21 - Execução do exemplo alterado de subprogramas com variável global. Fonte: O autor (2020)

Percebe-se que o **print()** do programa principal está na **linha 16**, depois da chamada à função **func2(x)**. Dessa forma, a variável global **x** foi alterada na execução da **func2(x)** e fica com o valor 20 quando a execução volta ao programa principal.

SUBPROGRAMAS ANINHADOS

Em Python, e na maioria das linguagens funcionais, é permitido aninhar subprogramas. Porém, as linguagens C e C++ não permitem essa prática. Veja o exemplo da Figura 22:

```

1 def taximetro(distancia)::
2 def calculaMult():
3 if distancia < 5:
4 return 1.2
5 else:
6 return 1
7 multiplicador = calculaMult()
8 largada = 3
9 km_rodado = 2
10 valor = (largada + distancia * km_rodado) * multiplicador
11 return valor
12
13
14 dist = eval(input("Entre com a distancia a ser percorrida em km: "))
15 pagamento = taximetro(dist)
16 print(f'O valor a pagar é R$ {pagamento}')
```

Figura 22 - Exemplo de subprogramas aninhados. Fonte: O autor (2020))

A função **taximetro()** tem, dentro de sua definição, a definição de outra função denominada **calculaMult()**. Na **linha 7**, a função **calculaMult()** é chamada e o seu retorno é armazenado na variável **multiplicador**.

MÉTODOS DE PASSAGENS DE PARÂMETROS

Os métodos de passagem de parâmetros são as maneiras que existem para transmiti-los ou recebê-los dos subprogramas chamados. Os parâmetros podem ser passados principalmente por:

VALOR

REFERÊNCIA

VALOR

O parâmetro formal funciona como uma variável local do subprograma, sendo inicializado com o valor do parâmetro real. Dessa maneira, não ocorre alteração na variável externa ao subprograma, caso ela seja passada como parâmetro.

REFERÊNCIA

Em vez de passar o valor do parâmetro real, é transmitido um caminho de acesso (normalmente um endereço) para o subprograma chamado. Isso fornece o caminho de acesso para a célula que armazena o parâmetro real. Assim, o subprograma chamado pode acessar o parâmetro real na unidade de programa chamadora.

SAIBA MAIS

Na linguagem C, utilizamos ponteiros para fazer a passagem de parâmetros por referência. As transmissões de parâmetros que não sejam ponteiros utilizam a passagem por valor.

O método de passagem de parâmetros de Python é chamado **passagem por atribuição**. Como todos os valores de dados são objetos, toda variável é uma referência para um objeto. Ao se estudar orientação a objetos, fica mais clara a diferença entre a passagem por atribuição e a passagem por referência. **Por enquanto, podemos entender que a passagem por atribuição é uma passagem por referência, pois os valores de todos os parâmetros reais são referências.**

RECURSIVIDADE

Uma função recursiva é aquela que chama a si mesma. Veja o exemplo da função **regressiva()**, como mostrado na Figura 23:

```
1 def regressiva(x):  
2 print(x)  
3 regressiva(x - 1)
```

Figura 23 - Exemplo de função recursiva - Fonte: O autor (2020)

Na implementação da função **regressiva()**, tendo **x** como parâmetro, ela própria é chamada com o parâmetro **x - 1**. Vamos analisar a chamada **regressiva(2)**:

Ao chamar **regressiva(2)**, o valor 2 é exibido na tela pela **linha 2**, e ocorre uma nova chamada da função **regressiva()** na **linha 3**, com o parâmetro 1. Vamos continuar com esse caminho de execução da **regressiva(1)**.



Ao chamar **regressiva(1)**, o valor 1 é exibido na tela pela **linha 2**, e ocorre uma nova chamada da função **regressiva()** na **linha 3**, com o parâmetro 0.



Ao chamar **regressiva(0)**, o valor 0 é exibido na tela e ocorre uma nova chamada da função **regressiva**, com o parâmetro -1, e assim sucessivamente.

ATENÇÃO

Conceitualmente, essa execução será repetida indefinidamente até que haja algum erro por falta de memória. Perceba que não definimos adequadamente uma condição de parada para a função **regressiva()**, o que leva a esse comportamento ruim.

Em Python, o interpretador pode interromper a execução indefinida, mas essa não é uma boa prática. Uma forma de contornar esse problema é definir adequadamente uma condição de parada, como no exemplo da Figura 24:

```
1 def regressiva(x):  
2 if x <= 0:  
3 print("Acabou")  
4 else:
```

5 print(x)

6 regressiva(x-1)

Figura 24 - Função recursiva com condição de parada - Fonte: O autor (2020)

Uma função recursiva que termina tem:

Um ou mais casos básicos, que funcionam como condição de parada da recursão.

Uma ou mais chamadas recursivas, que têm como parâmetros valores mais próximos do(s) caso(s) básico(s) do que o ponto de entrada da função.

Alguns exemplos clássicos de funções que podem ser implementadas de forma recursiva são o **cálculo do fatorial de um inteiro não negativo** e a **sequência de Fibonacci**, que serão explorados a seguir.

A FUNÇÃO RECURSIVA FATORIAL

A função matemática fatorial de um inteiro não negativo n é calculada por:

$$n! = \begin{cases} 1, & \text{se } n = 0 \text{ ou } n = 1 \\ n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1, & \text{se } n \geq 2 \end{cases}$$

Atenção! Para visualização completa da equação utilize a rolagem horizontal

Além disso, ela pode ser definida recursivamente por:

$$n! = \begin{cases} 1, & \text{se } n = 0 \text{ ou } n = 1 \\ n \cdot [(n-1)!], & \text{se } n \geq 2 \end{cases}$$

Atenção! Para visualização completa da equação utilize a rolagem horizontal

Uma implementação recursiva da função fatorial em Python está na Figura 25:

1 def fatorial(n):

2 if n == 0 or n == 1:

3 return 1

```
4 else:  
5 return n*fatorial(n-1)
```

Figura 25 - Função recursiva fatorial - Fonte: O autor (2020)

Vale ressaltar que a função fatorial também poderia ter sido implementada de forma **não recursiva**, como mostrado na Figura 26:

```
1 def fatorial(n):  
2 fat = 1  
3 if n == 0 or n == 1:  
4 return fat  
5 else:  
6 for x in range(2, n + 1):  
7 fat = fat*x  
8 return fat
```

Figura 26 - Função fatorial com laço for - Fonte: O autor (2020)

Porém, neste tópico, o intuito principal é explorar a recursividade.

A SEQUÊNCIA DE FIBONACCI

A sequência de Fibonacci é: 1, 1, 2, 3, 5, 8, 13, 21... Os dois primeiros termos são 1 e, a partir do 3º termo, cada termo é a soma dos dois anteriores.

Uma possível implementação recursiva de função que determina o n-ésimo termo da sequência de Fibonacci está na Figura 27:

```
1 def fibo(n):  
2 if n == 1 or n == 2:  
3 return 1  
4 else:  
5 return fibo(n - 1) + fibo(n - 2)
```

Figura 27 - Função recursiva para sequência de Fibonacci - Fonte: O autor (2020)

A **linha 2** traz as condições de parada.

A **linha 5** traz as chamadas recursivas para calcular os dois termos anteriores da sequência.

DOCSTRINGS

Em Python, é possível definir uma **string** que serve como documentação de funções definidas pelo desenvolvedor. Ao chamar o utilitário **help()** passando como parâmetro a função desejada, essa **string** é exibida. Veja a Figura 28 e a Figura 29:

```
1 def fibo(n):  
2 'Determina o n-ésimo termo da sequência de Fibonacci'  
3 if n == 1 or n == 2:  
4 return 1  
5 else:  
6 return fibo(n - 1) + fibo(n - 2)  
7  
8 print(help(fibo))
```

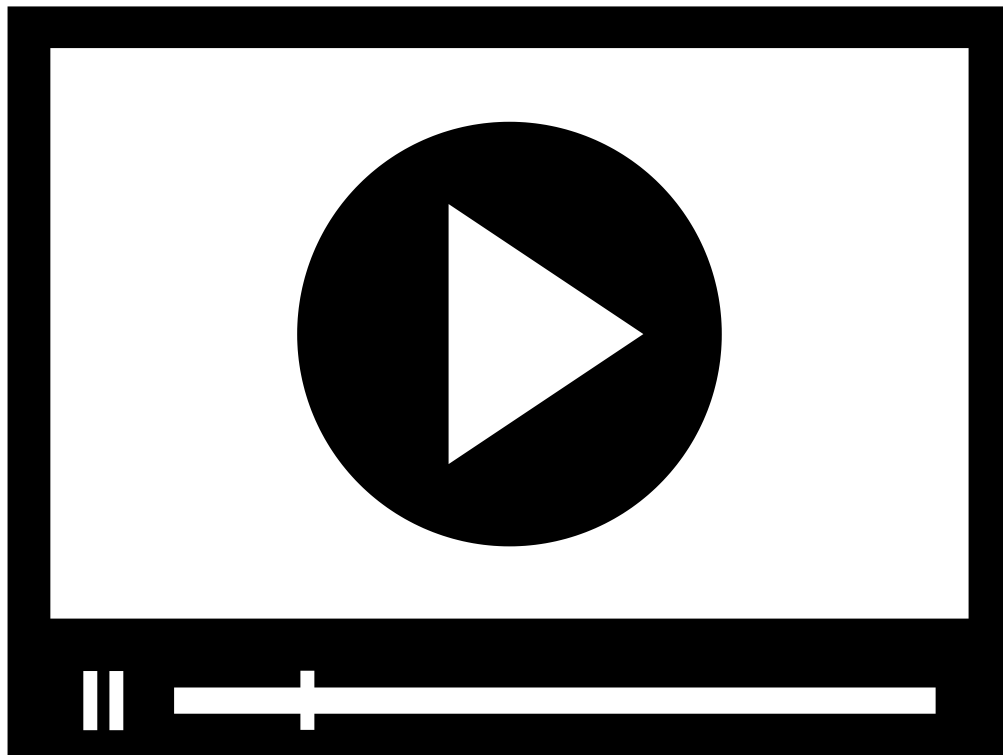
Figura 28 - Docstring da função fibo() - Fonte: O autor (2020)

```
1 fibo(n)  
2 Determina o n-ésimo termo da sequência de Fibonacci
```

Figura 29 - Exibição da docstring da função fibo() - Fonte: O autor (2020)

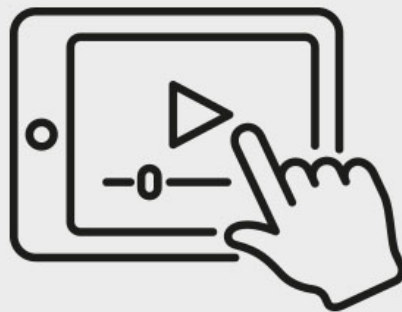
Na Figura 28, a **linha 2** mostra a declaração da docstring.

A **linha 8** mostra a impressão na tela da chamada **help(fibo)**. Na Figura 29, está o resultado da execução desse programa.



No vídeo a seguir, o professor nos apresenta exemplos práticos do uso de procedimentos e funções. Vamos assistir!

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. CONSIDERE O SEGUINTE TRECHO DE UM PROGRAMA ESCRITO EM PYTHON:

1DEF FUNC1(X):

2X = 10

3PRINT(X)

4

5

6X = 0

7PRINT(X)

8FUNC1(X)

9PRINT(X)

O QUE ACONTECERÁ QUANDO O USUÁRIO TENTAR EXECUTAR ESSE PROGRAMA?

A) Ocorrerá um erro e o programa não será executado.

B) Ocorrerá um erro durante a execução.

C) Será impresso na tela: 0 10 0

D) Será impresso na tela: 0 10 10

2. CONSIDERE O SEGUINTE TRECHO DE UM PROGRAMA, COM UMA IMPLEMENTAÇÃO DE FUNÇÃO RECURSIVA, ESCRITO EM PYTHON:

1DEF REC(N):

2IF N < 2:

3RETURN REC(N - 1)

4

5

6PRINT(REC(1))

QUANDO O USUÁRIO TENTOU EXECUTAR ESSE PROGRAMA, HOUVE UM ERRO. QUAL É A CAUSA?

☐ A) Na linha 2, o if está escrito de maneira errada.

B) A função não tem condição de parada.

C) A função está sem retorno.

D) A função não poderia ter sido definida com uma chamada a ela própria.

GABARITO

1. Considere o seguinte trecho de um programa escrito em Python:

```
1def func1(x):
```

```
2x = 10
```

```
3print(x)
```

```
4
```

```
5
```

```
6x = 0
```

```
7print(x)
```

```
8func1(x)
```

```
9print(x)
```

O que acontecerá quando o usuário tentar executar esse programa?

A alternativa **"C "** está correta.

A variável **x** da linha 6 é global. Mas, como existe outra variável com o mesmo nome dentro da função **func1()** – na linha 2, apenas dentro da função **func1()**, **x** vale 10 –, chamamos essa variável de local. Assim, o print da linha 7 recebe o valor da variável global (0). A execução da linha 8 chama a função **func1()**, que imprime o valor de **x** válido dentro dela (10). Em seguida, a execução do programa sai da função **func1()** e o print da linha 9 volta a enxergar a variável global **x**, cujo valor é 0.

2. Considere o seguinte trecho de um programa, com uma implementação de função recursiva, escrito em Python:

```
1def rec(n):
```

```
2if n < 2:
```

```
3return rec(n - 1)
```

4

5

```
6print(rec(1))
```

Quando o usuário tentou executar esse programa, houve um erro. Qual é a causa?

A alternativa "B " está correta.

A função é recursiva, mas não apresenta parada. Ao ser chamada com o parâmetro 1, o **if** da linha 2 tem condição verdadeira. Então, ocorre a chamada a `rec(0)`. Mas `rec(0)` não é definido e ocorrerá a chamada a `rec(-1)`. E assim sucessivamente.

MÓDULO 3

- 🕒 Identificar o uso correto de recursos de bibliotecas em Python



📷 Fonte: dTosh | Shutterstock

Python oferece, em seu núcleo, algumas funções que já utilizamos, como **print()** e **input()**, além de classes como **int**, **float** e **str**. Logicamente, o núcleo da linguagem Python disponibiliza muitas outras funções (ou métodos) e classes além das citadas. Mas, ainda assim, ele é pequeno, com objetivo de simplificar o uso e ganhar eficiência. Para aumentar a disponibilidade de funções, métodos e classes, o desenvolvedor pode usar a **biblioteca padrão Python**. Neste módulo, apresentaremos alguns dos principais recursos dessa biblioteca e a forma de utilizá-los.

BIBLIOTECA PADRÃO PYTHON

A biblioteca padrão Python consiste em milhares de funções, métodos e classes relacionados a determinada finalidade e organizados em componentes chamados módulos. São mais de 200 módulos que dão suporte, entre outras coisas, a:

Operações matemáticas.

Interface gráfica com o usuário (GUI).

Funções matemáticas e geração de números pseudoaleatórios.

ATENÇÃO

É importante lembrar dos conceitos de classes e objetos, pois eles são os principais conceitos do paradigma de programação orientada a objeto. As classes são fábricas, que podem gerar instâncias chamadas objetos. Uma classe **Pessoa**, por exemplo, pode ter como atributos nome e CPF. Ao gerar uma instância de **Pessoa**, com nome João da Silva e CPF 000.000.000-00, temos um objeto.

SAIBA MAIS

Para melhor compreensão dos conceitos de classe e objeto, pesquise sobre paradigma orientado a objeto.

COMO USAR UMA FUNÇÃO DE MÓDULO IMPORTADO

Para usar as funções e os métodos de um módulo, são necessários dois passos:

Fazer a importação do módulo desejado com a instrução:

import nome_modulo



Chamar a função desejada, precedida do nome do módulo, com a instrução:

nome_modulo.nome_funcao(paramêtros)

Como exemplo, vamos importar o módulo **math** (dedicado a operações matemáticas) e calcular a raiz quadrada de 5, por meio da função **sqrt()**. Observe a Figura 30:

```
>>> import math
>>> math.sqrt(5)
2.23606797749979
```

Figura 30 - Exemplo de uso do módulo math - Fonte: O autor (2020)

A partir desse ponto, serão apresentados os principais aspectos dos seguintes módulos:

MATH

usado para operações matemáticas.

RANDOM

usado para gerar números pseudoaleatórios.

SMTPLIB

usado para permitir envio de e-mails.

TIME

usado para implementar contadores temporais.

TKINTER

usado para desenvolver interfaces gráficas.

MÓDULO MATH

Esse módulo provê acesso a funções matemáticas de argumentos reais. As funções não podem ser usadas com números complexos.

O módulo **math** tem as funções listadas na Tabela 7, entre outras:

Função	Retorno
sqrt(x)	Raiz quadrada de x
ceil(x)	Menor inteiro maior ou igual a x
floor(x)	Maior inteiro menor ou igual a x
cos(x)	Cosseno de x
sin(x)	Seno de x
log(x, b)	Logaritmo de x na base b
pi	Valor de Pi (3.141592...)
e	Valor de e (2.718281...)

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Tabela 7 - Principais funções do módulo math - Fonte: O autor (2020)

SAIBA MAIS

Para mais informações sobre o módulo **math**, visite a biblioteca Python.

MÓDULO RANDOM

Esse módulo implementa geradores de números pseudoaleatórios para várias distribuições.

NÚMEROS INTEIROS

SEQUÊNCIAS

NÚMEROS INTEIROS

Para inteiros, existe:

Uma seleção uniforme a partir de um intervalo.

SEQUÊNCIAS

Para sequências, existem:

Uma seleção uniforme de um elemento aleatório;

Uma função para gerar uma permutação aleatória das posições na lista;

Uma função para escolher aleatoriamente sem substituição.

DISTRIBUIÇÕES DE VALORES REAIS

A Tabela 8 mostra algumas das principais funções disponíveis para distribuições de valores reais no módulo **random**.

Função	Retorno
random()	Número de ponto flutuante no intervalo [0.0, 1.0)
uniform(a, b)	Número de ponto flutuante N tal que $a \leq N \leq b$
gauss(mu, sigma)	Distribuição gaussiana. mu é a média e sigma é o desvio padrão.
normalvariate(mu, sigma)	Distribuição normal. mu é a média e sigma é o desvio padrão.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Tabela 8 - Principais distribuições de valores reais - Fonte: O autor (2020)

FUNÇÕES PARA NÚMEROS INTEIROS

A Tabela 9 mostra algumas das principais funções disponíveis para inteiros no módulo **random**.

Função	Retorno

randrange(stop)	Um elemento selecionado aleatório de range (start, stop, step), mas sem construir um objeto range .
randrange(start, stop [,step])	
randint(a, b)	Número inteiro N tal que $a \leq N \leq b$

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Tabela 9 - Principais funções do módulo random para inteiros - Fonte: O autor (2020)

FUNÇÕES PARA SEQUÊNCIAS

A Tabela 10 mostra algumas das principais funções disponíveis para sequências no módulo **random**.

Função	Retorno
choice(seq)	Elemento aleatório de uma sequência não vazia seq .
shuffle(x[, random])	Embaralha a sequência x no lugar.
sample(pop, k)	Uma sequência de tamanho k de elementos escolhidos da população pop , sem repetição. Usada para amostragem sem substituição.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Tabela 10 - Principais funções do módulo random para sequências - Fonte: O autor (2020)

Para mais informações sobre o módulo **random**, visite a biblioteca Python.

MÓDULO SMTPLIB

Esse módulo define um objeto de sessão do cliente **SMTP** que pode ser usado para enviar e-mail para qualquer máquina da internet com um serviço de processamento **SMTP** ou **ESMTP**. O exemplo a seguir vai permitir que você envie um e-mail a partir do servidor SMTP do Gmail.

Como a Google não permite, por padrão, realizar o login com a utilização do **smtplib** por considerar esse tipo de conexão mais arriscada, será necessário alterar uma configuração de segurança. Para resolver isso, siga as instruções a seguir:

Acesse sua conta no Google.



Depois **Segurança**.



Acesso a app menos seguro.



Mude para "ativada" a opção de "Permitir aplicativos menos seguros".

ACESSO A APP MENOS SEGURO

Atenção!

Em algumas contas, os nomes estarão escritos no idioma inglês (*Allow less secure apps*).

Para fazer seu primeiro envio, crie um programa novo no seu projeto. A Figura 31 mostra as importações necessárias para o envio:

```
1 #import dos pacotes necessários
2 from email.mime.multipart import MIMEMultipart
3 from email.mime.text import MIMEText
4 import smtplib
5
```

Figura 31 - Envio de e-mail com módulo smtplib 1 - Fonte: O autor (2020)

A Figura 32 mostra a criação da mensagem, com o corpo e seus parâmetros.

```
6 #criação de um objeto de mensagem
7 msg = MIMEMultipart()
8 texto = "Estou enviando um email com Python"
9
10 #parâmetros
11 senha = "SUA SENHA"
12 msg['From'] = "SEU E-MAIL"
13 msg['To'] = "E-MAIL DESTINO"
14 msg['Subject'] = "ASSUNTO"
15
16 #criação do corpo da mensagem
17 msg.attach(MIMEText(texto, 'plain'))
18
```

Figura 32 - Envio de e-mail com módulo smtplib 2 - Fonte: O autor (2020)

A **linha 7** mostra a criação de um objeto de mensagem.

A **linha 8** mostra o corpo da mensagem em uma **string**.

As **linhas de 11 a 14** devem ser preenchidas com os valores adequados para que seu programa seja executado com sucesso.

A **linha 17** anexa o corpo da mensagem (que estava em uma **string**) ao objeto **msg**.

A Figura 33 mostra os passos necessários para o envio em si.

```
19 #criação do servidor
20 server = smtplib.SMTP('smtp.gmail.com: 587')
21 server.starttls()
```

```
22
23 #Login na conta para envio
24 server.login(msg['From'], senha)
25
26 #envio da mensagem
27 server.sendmail(msg['From'], msg['To'], msg.as_string())
28
29 #encerramento do servidor
30 server.quit()
31
32 print('Mensagem enviada com sucesso')
33
```

Figura 33 - Envio de e-mail com módulo `smtplib` 3 - Fonte: O autor (2020)

As **linhas 20 e 21** mostram a criação do servidor e a sua conexão no modo TLS.

A **linha 24** mostra o login na conta de origem do e-mail.

A **linha 27** representa o envio propriamente dito.

A **linha 30** mostra o encerramento do servidor.

SAIBA MAIS

Para mais informações sobre o módulo **`smtplib`**, visite a biblioteca Python.

MÓDULO TIME

Esse módulo provê diversas funções relacionadas a tempo. Também pode ser útil conhecer os módulos **`datetime`** e **`calendar`**. A Tabela 11 mostra algumas das principais funções disponíveis no módulo **`time`**.

Função	Retorno
<code>time()</code>	Número de segundos passados desde o início da contagem (epoch). Por padrão, o início é 00:00:00 do dia 1 de janeiro de 1970.
<code>ctime(segundos)</code>	Uma string representando o horário local, calculado a partir do número de segundos passado como parâmetro.
<code>gmtime(segundos)</code>	Converte o número de segundos em um objeto struct_time descrito a seguir.
<code>localtime(segundos)</code>	Semelhante à gmtime() , mas converte para o horário local.
<code>sleep(segundos)</code>	A função suspende a execução por determinado número de segundos.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Tabela 11 - Principais funções do módulo time - Fonte: O autor (2020)

A Figura 34 mostra um exemplo de chamada das funções **time()** e **ctime()**.

```
>>> x = time.time()
>>> print(f'Local time: {time.ctime(x)}')
```

Local time: Tue Jun 30 23:38:55 2020

Figura 34 - Exemplo de uso das funções time() e ctime() - Fonte: O autor (2020)

A variável **x** recebe o número de segundos desde 00:00:00 de 01/01/1970 pela função **time()**. Ao executar **ctime(x)**, o número de segundos armazenado em **x** é convertido em uma **string**, com o horário local.

A classe **time.struct_time** gera objetos sequenciais com valor de tempo retornado pelas funções **gmtime()** e **localtime()**. São objetos com interface de **tupla** nomeada: os valores podem ser acessados pelo índice e pelo nome do atributo. Existem os seguintes valores que estão apresentados na Tabela 12:

TUPLA

Tupla é uma sequência finita de objetos. Uma tupla é similar à estrutura de dados lista, porém, a principal diferença é que uma tupla, após definida, não permite a adição ou remoção de elementos.

Índice	Atributo	Valores
0	tm_year	Por exemplo, 2020
1	tm_mon	range [1, 12]
2	tm_mday	range [1, 31]
3	tm_hour	range [0, 23]
4	tm_min	range [0, 59]
5	tm_sec	range [0, 61]
6	tm_wday	range [0, 6], Domingo é 0
7	tm_yday	range [1, 366]
8	tm_isdst	0, 1 ou -1
N/A	tm_zone	abreviação do nome da timezone

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

SAIBA MAIS

Para mais informações sobre o módulo **time**, visite a biblioteca Python.

MÓDULO TKINTER

O pacote **tkinter** é a interface Python padrão para o Tk GUI (interface gráfica com o usuário) *toolkit*. Na maioria dos casos, basta importar o próprio **tkinter**, mas diversos outros módulos estão disponíveis no pacote. A biblioteca **tkinter** permite a criação de janelas com elementos gráficos, como entrada de dados e botões, por exemplo.

O exemplo a seguir vai permitir que você crie a primeira janela com alguns elementos. Para isso, crie um programa novo no seu projeto. A Figura 35 mostra a criação da sua primeira janela, ainda sem qualquer elemento gráfico.

```
1 from tkinter import *  
2  
3 janelaPrincipal = Tk()  
4 janelaPrincipal.mainloop()
```

Figura 35 - Primeira janela com tkinter 1 - Fonte: O autor (2020)

A **linha 1** mostra a importação de todos os elementos disponíveis em **tkinter**. O objeto **janelaPrincipal** é do tipo Tk. Um objeto Tk é um elemento que representa a janela GUI. Para que essa janela apareça, é necessário chamar o método **mainloop()**;

Para exibir textos, vamos usar o elemento **Label**. A Figura 36 mostra as **linhas 4 e 5**, com a criação do elemento e o seu posicionamento. O tamanho padrão da janela é 200 X 200 pixels, com o canto superior esquerdo de coordenadas (0,0) e o canto inferior direito de coordenadas (200,200).

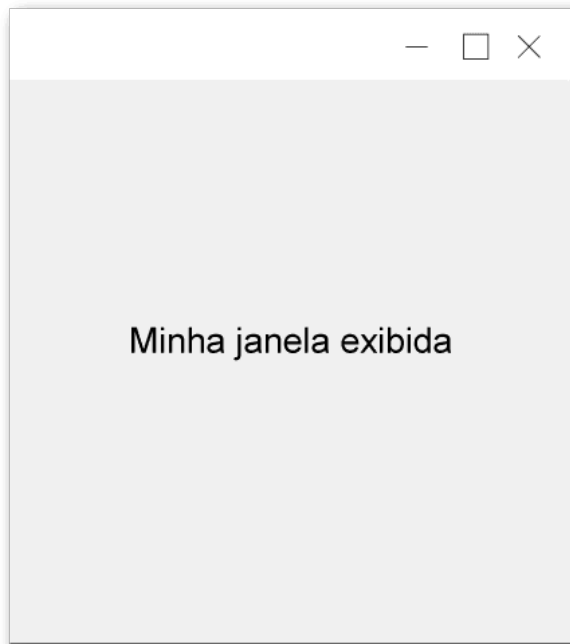
```
1 from tkinter import *
```

```
2
```

```
3 janelaPrincipal = Tk()
4 texto = Label(master = janelaPrincipal, text = "Minha janela exibida")
5 texto.place(x = 50 y = 100)
6 janelaPrincipal.mainloop()
```

Figura 36 - Primeira janela com tkinter 2 - Fonte: O autor (2020)

Veja o resultado de sua primeira janela, apenas com o texto, na Figura 37.



Fonte:Shutterstock

Figura 37 - Exibição da primeira janela com tkinter - Fonte: O autor (2020)

Vamos agora incrementar um pouco essa janela. Para isso, vamos acrescentar uma imagem e um botão. **A imagem precisa estar na mesma pasta do seu arquivo .py.**

```
1 from tkinter import *
2
3 def funcClicar():
4     print("Botão pressionado")
5
6 janelaPrincipal = Tk()
7 texto = Label(master = janelaPrincipal, text = "Minha janela exibida")
8 texto.pack()
9
10 pic = PhotoImage(file="logoEstacio.gif")
```

```
11 logo = Label(master = janelaPrincipal, image = pic)
12 logo.pack()
13
14 botao = Button(master = janelaPrincipal, text = 'Clique', command = funcClicar)
15 botao.pack()
16
17 janelaPrincipal.mainloop()
18
```

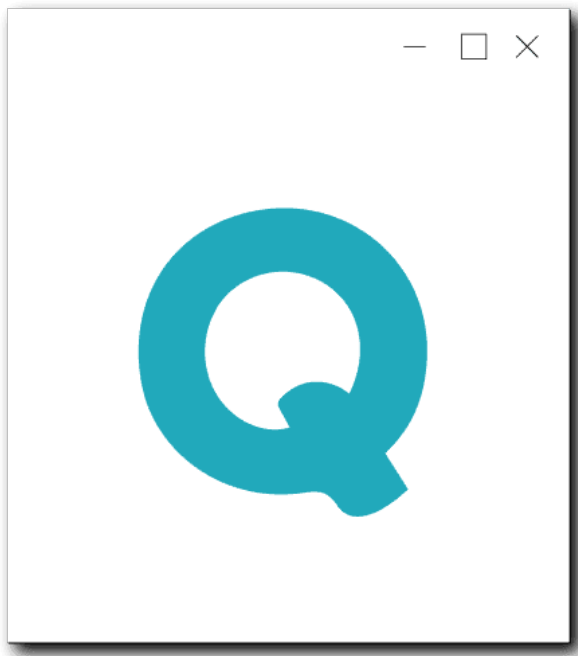
Figura 38 - Segunda janela com tkinter - Fonte: O autor (2020)

Na Figura 38, temos a inserção do elemento de imagem e o botão. Nas **linhas 10, 11 e 12**, é feita a criação do objeto **Label** para conter a imagem e seu posicionamento. Observe que passamos a utilizar o método **pack()**, que coloca o elemento centralizado e posicionado o mais perto possível do topo, depois dos elementos posicionados anteriormente.

O elemento botão é criado na **linha 14**, com os atributos **text** e **command**, que, respectivamente, são o texto exibido no corpo do botão e a função a ser executada quando o botão for clicado.

Para o funcionamento correto do botão, precisamos definir a função **funcClicar()**, nas **linhas 3 e 4**. Essa função serve apenas para imprimir na tela a **string** “Botão pressionado”.

Veja o resultado na Figura 39 e, depois de implementar sua própria janela, clique no botão para ver o resultado no console, como na Figura 40.



Fonte: Shutterstock

Figura 39 - Segunda janela com tkinter exibida - Fonte: O autor (2020)

Botão pressionado

Figura 40 - Resultado do clique no botão da segunda janela - Fonte: O autor (2020)

+ SAIBA MAIS

Para mais informações sobre o **tkinter**, visite a biblioteca Python.

USANDO PACOTES EXTERNOS

Além dos módulos fornecidos de forma integrada pela biblioteca padrão do Python, a linguagem possui uma grande vantagem: sua característica *open-source* permite que qualquer desenvolvedor, com o conhecimento adequado, desenvolva sua própria biblioteca e os seus próprios módulos, que chamaremos a partir de agora de **pacotes**.

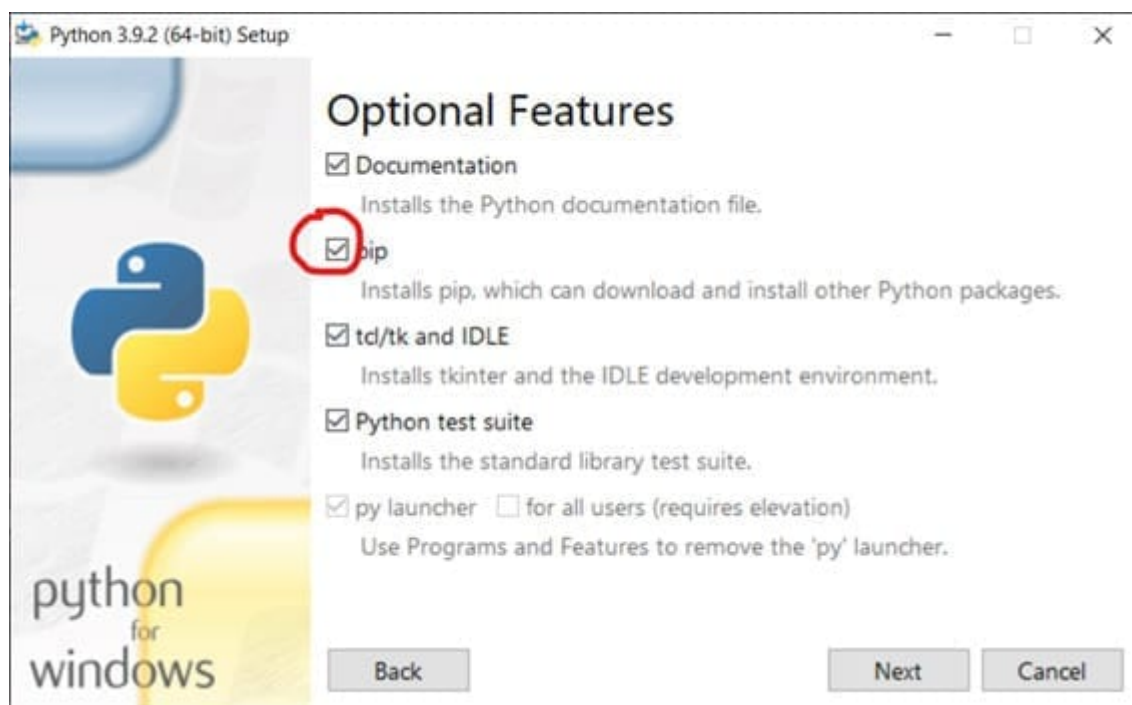
💡 DICA

Veremos como criar módulos mais adiante neste conteúdo.

Com um pacote pronto, o desenvolvedor pode disponibilizar esse material na internet, de forma que qualquer pessoa possa aproveitar o código implementado. Isso foi um dos fatores que fez com que o Python se tornasse uma das linguagens mais utilizadas no mundo atualmente.

Como forma de facilitar a distribuição dos pacotes entre os usuários, existe um grupo dentro da comunidade Python que mantém o chamado *Python Package Index*, ou PyPI, que é um grande servidor no qual os desenvolvedores podem hospedar os seus pacotes, contendo bibliotecas e/ou módulos para que sejam baixados e instalados por outras pessoas.

É possível acessar o PyPI através do **pip**, um programa que pode ser instalado juntamente com a distribuição do Python. Para isso, certifique-se de que a caixa “pip” está marcada durante a instalação, conforme ilustrado a seguir.



📷 Instalação do pip.

💡 DICA

Para verificar se a sua instalação Python incluiu o pip, procure pela pasta **Scripts** dentro do diretório de instalação que você escolheu para o Python. Dentro dessa pasta, localize o arquivo

pip.exe. Caso não o encontre, pesquise sobre como instalar o **pip**. O mais recomendado é tentar reinstalar o Python, lembrando de marcar a opção de incluir o pip durante o processo de instalação.

Além do pip, é necessário termos o endereço para acessar o pip dentro da variável de ambiente **PATH**. O processo de instalação do python também permite incluir automaticamente o Python no seu **PATH**, porém, caso não tenha feito isso, siga os passos abaixo:

1

Clique na **tecla do Windows** e escreva “*Editar as variáveis de ambiente para a sua conta*”.

Na janela que abrir, procure pela variável **Path**, selecione-a e clique no botão “**Editar**”.

2

3

Clique no botão “**Novo**” e digite o endereço da sua instalação do Python (por exemplo, *D:\Python*).

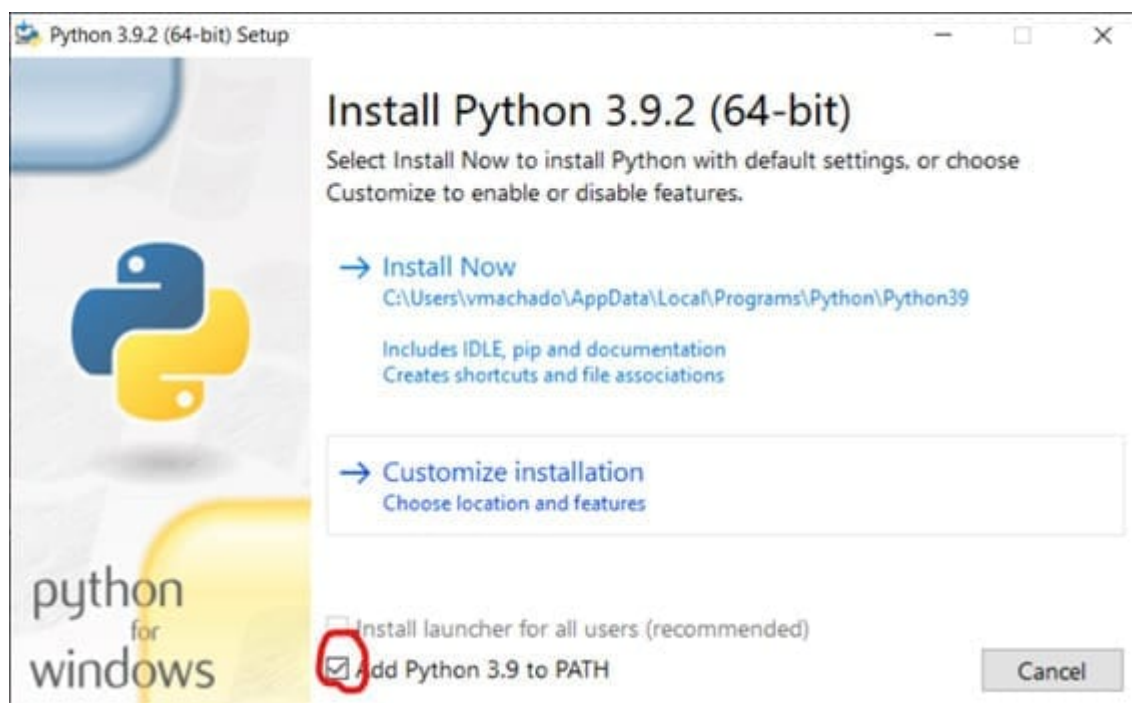
Clique no botão “**Novo**” uma segunda vez e digite o endereço da pasta **Scripts** dentro da sua instalação do Python (por exemplo, *D:\Python\Scripts*).

4

5

Aperte **Ok** até fechar todas as janelas.

É importante que você se certifique de que a opção “*Add Python to PATH*” está marcada durante a instalação, como ilustrado a seguir.



📷 Adicionando o Python ao Path.

Com essas etapas concluídas, já conseguimos instalar nossos pacotes externos!

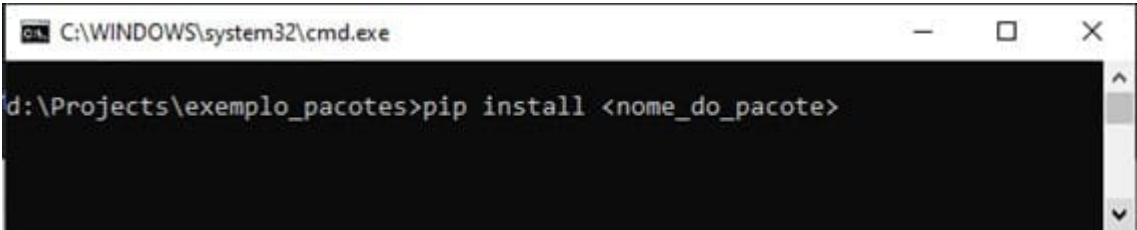
📢 ATENÇÃO


Quando estiver trabalhando com pacotes externos, é extremamente recomendado o uso de ambientes virtuais (em inglês *virtual environments* ou simplesmente *virtualenvs*). Esses ambientes isolam o projeto em que você está trabalhando, e uma das vantagens disso é que você consegue saber exatamente quais pacotes externos estão sendo usados no projeto.

É possível usar pacotes externos sem o uso de ambientes virtuais, porém isso pode causar uma confusão caso você tenha vários projetos Python no seu computador.

Pesquise mais sobre ambientes virtuais e configure um em cada projeto. Não é muito difícil e vai te ajudar a deixar o seu código mais profissional!

Para instalar um pacote externo disponível no PyPI, basta abrir o seu terminal (clique no botão do Windows e digite “cmd” ou “prompt de comando”), ative o seu ambiente virtual (se você estiver usando um) e digite o seguinte comando: `d: \Projects\exemplo_pacotes>pip install <nome_do_pacote>`



 Instalando um pacote usando pip.

Substitua `<nome_do_pacote>` pelo pacote que você deseja usar. Temos inúmeros pacotes prontos à nossa disposição. Cada pacote normalmente possui um site que apresenta a sua documentação, de forma similar à documentação oficial do Python.

Abaixo você encontra uma lista com alguns dos pacotes externos mais comuns e utilizados no mercado:

Nome do módulo	Para que serve?
numpy	Cálculos, operações matemáticas e simulações
pandas	Manipulação de dados
scikit-learn	Modelos de aprendizado de máquina
matplotlib	Visualização de dados
requests	Biblioteca de comandos de comunicação pelo protocolo HTTP
flask	Construção de aplicações web

Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal

 Pacotes externos mais comuns e utilizados no mercado.

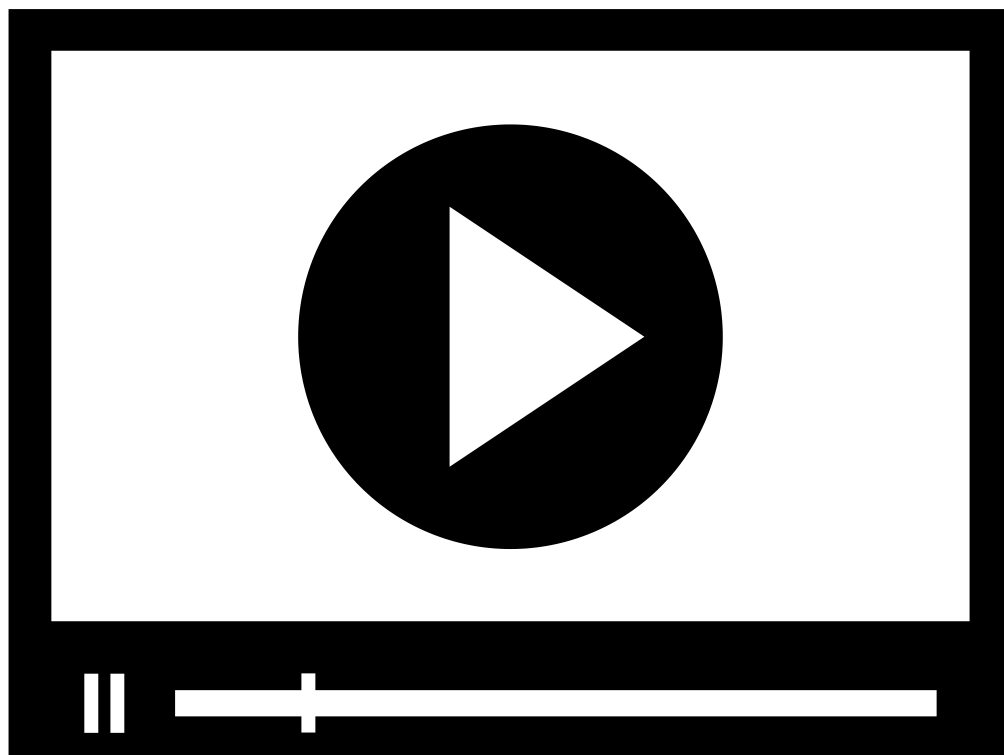
💡 DICA

Antes de começar o seu próprio módulo, é sempre recomendado pesquisar se o que você quer fazer já existe em algum pacote popular. Se existir, procure pela documentação e instale esse pacote.

O uso de módulos oriundos de pacotes externos é idêntico ao uso dos módulos da biblioteca padrão, basta utilizar o **import nome_do_modulo** no seu código.

CRIAÇÃO DO PRÓPRIO MÓDULO

Os desenvolvedores podem criar seus próprios módulos de forma a reutilizar as funções que já escreveram e organizar melhor seu trabalho. Para isso, basta criar um arquivo .py e escrever nele suas funções. **É importante observar que o arquivo do módulo precisa estar na mesma pasta do arquivo para onde ele será importado.**



No vídeo a seguir, o professor nos apresenta exemplos práticos do uso das bibliotecas de Python. Vamos assistir!

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. SABEMOS QUE É POSSÍVEL IMPORTAR MÓDULOS E CHAMAR FUNÇÕES DESSES MÓDULOS EM PYTHON. CONSIDERE O MÓDULO MATH, QUE OFERECE DIVERSAS FUNÇÕES MATEMÁTICAS. UMA DESSAS FUNÇÕES É A CEIL(X), QUE RETORNA O MENOR INTEIRO MAIOR OU IGUAL A X. SUPONHA QUE UM ESTUDANTE QUEIRA USAR UMA VARIÁVEL N, QUE RECEBE O VALOR 5.9, E EM SEGUIDA IMPRIMIR NA TELA O MENOR INTEIRO MAIOR OU IGUAL A ELA.

A)

```
1import math
```

```
2n = 5.9
```

```
3print(ceil(n))
```

A)

B)

```
1import math
```

```
2n = 5.9
```

```
3math.ceil(n)
```

```
4print(n)
```

B)

C)

```
1import math  
2n = 5.9  
3print(ceil.math(n))
```

C)

D)

```
1import math  
2n = 5.9  
3print(math.ceil(n))
```

D)

2. SOBRE A LINGUAGEM PYTHON E SUA BIBLIOTECA PADRÃO, É CORRETO AFIRMAR QUE:

- A) Só permite a utilização dos módulos contidos na biblioteca padrão Python.
- B) Tem o módulo de interface gráfica **tkinter**, que não permite a criação de janelas com botões.
- C) Tem módulo de interface de e-mails **smtplib**, que não permite envio de e-mails por servidores gratuitos.
- D) Tem módulo de operações matemáticas **math**, que não permite operações com números complexos.

GABARITO

1. Sabemos que é possível importar módulos e chamar funções desses módulos em Python. Considere o módulo **math**, que oferece diversas funções matemáticas. Uma dessas funções é a **ceil(x)**, que retorna o menor inteiro maior ou igual a **x**. Suponha que um estudante queira usar uma variável **n**, que recebe o valor **5.9**, e em seguida imprimir na tela o menor inteiro maior ou igual a ela.

A alternativa "D " está correta.

A maneira correta de usar uma função de um módulo importado é: (i) importar o módulo e (ii) chamar a função com a sintaxe **nomeModulo.nomeFuncao()**.

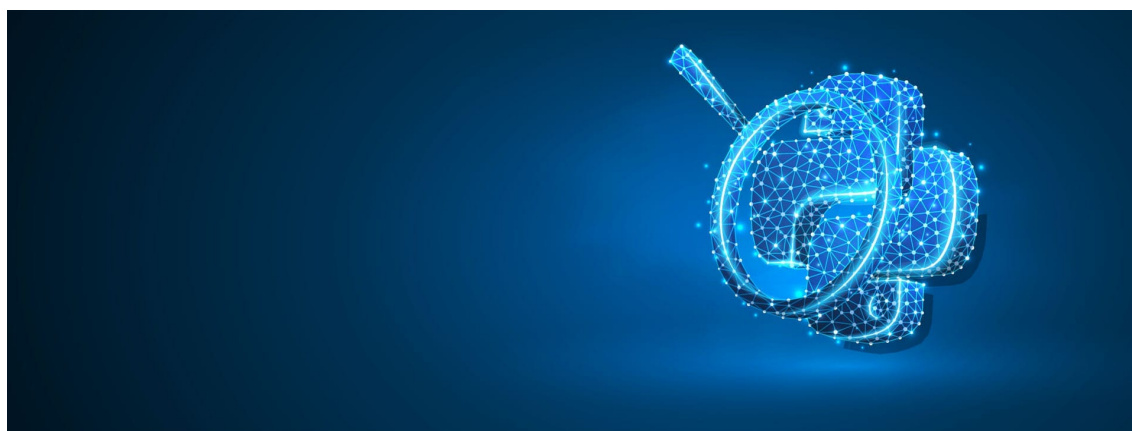
2. Sobre a linguagem Python e sua biblioteca padrão, é correto afirmar que:

A alternativa "D " está correta.

O módulo **math** não permite operações com números complexos.

MÓDULO 4

- ⦿ Analisar as formas de tratamento de exceções e eventos em Python.



📷 Fonte: dTosh | Shutterstock

Até agora, consideramos que nossos programas tiveram seu fluxo de execução normal. Neste módulo, vamos analisar o que acontece quando o fluxo de execução é interrompido por uma exceção, além de controlar esse fluxo excepcional.

ERROS E EXCEÇÕES

Dois tipos básicos de erros podem acontecer em um programa em Python. Os **erros de sintaxe** são aqueles que ocorrem devido ao formato incorreto de uma instrução. Esses erros são descobertos pelo componente do interpretador Python, chamado **analisador** ou **parser**.

Veja exemplos na Figura 41 e na Figura 42:

```
>>> print 'hello'
File "<input>", line 1
print 'hello'
^
```

SyntaxError: Missing parentheses in call to 'print'. Did you mean print('hello')?

Figura 41 - Erro de sintaxe 1 - Fonte: O autor (2020)

```
>>> lista = [1 ; 2 ; 4]
File "<input>", line 1
lista = [1 ; 2 ; 4]
^
```

SyntaxError: invalid syntax

Figura 42 - Erro de sintaxe 2 - Fonte: O autor (2020)

Além desses, existem os **erros que ocorrem em tempo de execução do programa**, que não são devidos a uma instrução escrita errada, mas sim ao fato de que o programa entrou em um estado indevido. Temos os exemplos:

A divisão por 0.

A tentativa de acessar um índice indevido em uma lista.

Um nome de variável não atribuído.

Um erro causado por tipos incorretos de operando.

Em cada caso, quando o programa atinge um estado inválido, dizemos que o interpretador Python **levanta uma exceção**. Isso significa que é criado um objeto que contém as informações relevantes sobre o erro. A Tabela 13 traz alguns tipos comuns de exceção:

Exceção	Explicação
KeyboardInterrupt	Levantado quando o usuário pressiona CTRL + C, a combinação de interrupção.
OverflowError	Levantado quando uma expressão de ponto flutuante é avaliada como um valor muito grande.
ZeroDivisionError	Levantado quando se tenta dividir por 0.

IOError	Levantado quando uma operação de entrada/saída falha por um motivo relacionado a isso.
IndexError	Levantado quando um índice sequencial está fora do intervalo de índices válidos.
NameError	Levantado quando se tenta avaliar um identificador (nome) não atribuído.
TypeError	Levantado quando uma operação da função é aplicada a um objeto do tipo errado.
ValueError	Levantado quando a operação ou função tem um argumento com o tipo correto, mas valor incorreto.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Tabela 13 - Tipos comuns de exceção - Extraído de Perkovic (2016)

Em Python, as exceções são objetos. A classe **Exception** é derivada de **BaseException**, classe base de todas as classes de exceção. **BaseException** fornece alguns serviços úteis para todas as classes de exceção, mas normalmente não se torna uma subclasse diretamente.

CAPTURA E MANIPULAÇÃO DE EXCEÇÕES

Para evitar que os programas sejam interrompidos quando uma exceção for levantada, é possível planejar um comportamento alternativo. Assim, o programa não será interrompido e a exceção poderá ser tratada. Chamamos esse processo de **captura da exceção**.

Vamos considerar um exemplo de programa que solicita ao usuário, com a função **input()**, um número inteiro. Embora essa função trate a entrada do usuário como **string**, é possível utilizá-la em conjunto com a função **eval()** para que os dados inseridos sejam avaliados como números. A Figura 43 mostra uma implementação simples desse exemplo.

```
1 num = eval(input("Entre com um número inteiro: "))
2 print(num)
```

Figura 43 - Exemplo simples de input - Fonte: O autor (2020)

Mas o que aconteceria se o usuário digitasse uma palavra em vez de números?

Veja a Figura 44.

Entre com um número inteiro: *dez*

Traceback (most recent call last):

File "C:/Users/Humberto/PycharmProjects/estacio_ead/excessoes.py", line 1, in <module>

```
num = eval(input("Entrecom um número inteiro: "))
```

File "<string>", line 1, in <module>

NameError: name 'dez'is not defined

Figura 44 - Exemplo de inserção de dados inválida sem tratamento - Fonte: O autor (2020)

Veja que o programa foi encerrado com uma exceção sendo levantada. Uma forma de fazer a captura e a manipulação de exceções é usar o par de instruções **try / except**.

O bloco **try** é executado primeiro, no qual devem ser inseridas as instruções do fluxo normal do programa.



O bloco **except** somente será executado se houver o levantamento de alguma exceção.

Isso permite que o fluxo de execução continue de maneira alternativa. A Figura 45 mostra uma implementação possível desse exemplo:

```
1 try:
2 num = eval(input("Entre com um número inteiro: "))
3 print(num)
4 except:
5 print("Entre com o valor numérico e não letras")
```

Figura 45 - Exemplo de uso de try/except - Fonte: O autor (2020)

A execução pode ser conferida na Figura 46:

Entre com um número inteiro: *dez*

Entre com valor numérico e não letras

O formato padrão de uso do par **try/except** é:

- 1 try:
- 2 Bloco 1
- 3 except:
- 4 Bloco 2
- 5 Instrução fora do try/except

O **bloco 1** representa o fluxo normal do programa. Caso uma exceção seja levantada, o **bloco 2** será executado, permitindo o tratamento adequado dela. Esse **bloco 2** é chamado de **manipulador de exceção**.

ATENÇÃO

Em Python, o manipulador de exceção padrão é quem executa o trabalho de captura da exceção caso não haja um tratamento explícito feito pelo desenvolvedor. É esse manipulador padrão o responsável pela exibição das mensagens de erro no console, como visto na Figura 44.

CAPTURA DE EXCEÇÕES DE DETERMINADO TIPO

Python permite que o bloco relativo ao **except** só seja executado caso a exceção levantada seja de determinado tipo. Para isso, o **except** precisa trazer o tipo de exceção que se deseja capturar. A Figura 47 traz uma possível variação do exemplo anterior, com a captura apenas das exceções do tipo **NameError**.

- 1 try:
- 2 num = eval(input("Entre com um número inteiro: "))
- 3 print(num)
- 4 except NameError:
- 5 print("Entre com o valor numérico e não letras")

CAPTURA DE EXCEÇÕES DE MÚLTIPLOS TIPOS

Python permite que haja diversos tratamentos para diferentes tipos possíveis de exceção. Isso pode ser feito com mais de uma cláusula **except** vinculada à mesma cláusula **try**. A Figura 48 mostra um exemplo de implementação da captura de exceções de múltiplos tipos.

```
1 try:
2 num = eval(input("Entre com um número inteiro: "))
3 print(num)
4 except ValueError:
5 print("Mensagem 1")
6 except IndexError:
7 print("Mensagem 2")
8 except:
9 print("Mensagem 3")
```

Figura 48 - Captura de múltiplos tipos de exceção - Fonte: O autor (2020)

A instrução da **linha 5** somente será executada se a exceção levantada no bloco **try** for do tipo **ValueError**, e se, na instrução da **linha 7**, a exceção for do tipo **IndexError**.

Caso a exceção seja de outro tipo, a **linha 9** será executada.

O TRATAMENTO COMPLETO DAS EXCEÇÕES

A forma geral completa para lidar com as exceções em Python é:

```
try:
2 Bloco 1
3 except Exception1:
4 Bloco tratador para Exception1
```

5 except Exception2:

6 Bloco tratador para Exception1

7 ...

8 else:

9 Bloco 2 – executado caso nenhuma exceção seja levantada

10 finally:

11 Bloco 3 – executado independente do que ocorrer

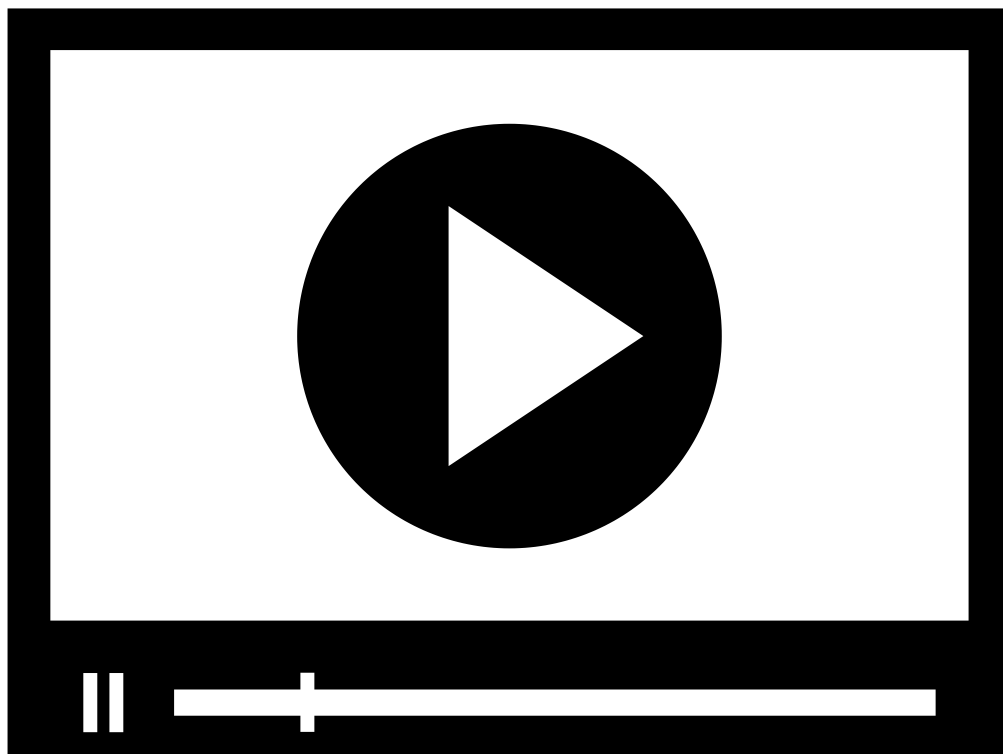
12 Instrução fora do try/except

As cláusulas **else** e **finally** são opcionais, como foi possível perceber nos exemplos iniciais.

TRATAMENTO DE EVENTOS

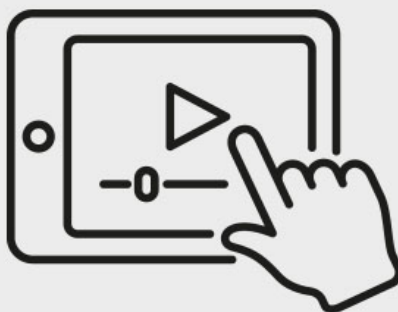
O tratamento de eventos é similar ao tratamento de exceções. Assim como podemos tratar as exceções ocorridas em tempo de execução, podemos tratar os eventos criados por ações externas, como interações de usuário realizadas por meio de uma interface gráfica de usuário (GUI).

UM EVENTO É A NOTIFICAÇÃO DE QUE ALGUMA COISA ACONTECEU, COMO UM CLIQUE DE MOUSE SOBRE UM ELEMENTO BOTÃO. O TRATADOR DO EVENTO É O SEGMENTO DE CÓDIGO QUE SERÁ EXECUTADO EM RESPOSTA À OCORRÊNCIA DO EVENTO.



No vídeo a seguir, o professor nos apresenta exemplos práticos do tratamento das exceções.
Vamos assistir!

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. CONSIDERE O SEGUINTE TRECHO DE UM PROGRAMA ESCRITO EM PYTHON:

1 TRY:

2 NUM = EVAL(INPUT("ENTRE COM UM NÚMERO INTEIRO: "))

3 PRINT(NUM)

4 EXCEPT VALUEERROR:

5 PRINT("MENSAGEM 1")

6 EXCEPT INDEXERROR:

7 PRINT("MENSAGEM 2")

8 EXCEPT:

9 PRINT("MENSAGEM 3")

SUPONHA QUE DURANTE A EXECUÇÃO O USUÁRIO ENTRE COM A PALAVRA NUMERO QUANDO SOLICITADO. ASSINALE A OPÇÃO QUE MOSTRA O RESULTADO IMEDIATO DESSA AÇÃO.

A) O programa deixará de ser executado.

B) Será impresso na tela **Mensagem 1**.

C) Será impresso na tela **Mensagem 2**.

D) Será impresso na tela **Mensagem 3**.

2. SOBRE O TRATAMENTO DE EXCEÇÕES EM PYTHON, É INCORRETO AFIRMAR QUE:

A) É possível implementar tratamentos diferentes de acordo com a exceção levantada.

B) Não é possível utilizar a cláusula **finally**.

C) Não é possível utilizar a cláusula **catch**.

D) É possível implementar um tratamento geral para todas as exceções levantadas.

GABARITO

1. Considere o seguinte trecho de um programa escrito em Python:

```
1 try:
2 num = eval(input("Entre com um número inteiro: "))
3 print(num)
4 except ValueError:
5 print("Mensagem 1")
6 except IndexError:
7 print("Mensagem 2")
8 except:
9 print("Mensagem 3")
```

Suponha que durante a execução o usuário entre com a palavra **numero** quando solicitado. Assinale a opção que mostra o resultado imediato dessa ação.

A alternativa "D " está correta.

Como o usuário inseriu uma palavra e não um número, a exceção não será do tipo **ValueError** nem do tipo **IndexError**. Assim, a cláusula **except** a ser executada é a da linha 8, imprimindo **Mensagem 3**.

2. Sobre o tratamento de exceções em Python, é incorreto afirmar que:

A alternativa "B " está correta.

A cláusula **finally** pode ser usada, embora não seja obrigatória.

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Neste tema, você aprendeu a usar as estruturas de controle, sejam de decisão ou de repetição, foi apresentado aos conceitos de subprogramas em Python, implementou suas próprias funções, além de conhecer e utilizar as bibliotecas. Por fim, analisou as formas de tratamento

de exceções e eventos. Com esse conteúdo, você com certeza terá condições de desenvolver aplicações muito mais complexas e com muito mais recursos.

Para ouvir um *podcast* sobre o assunto, acesse a versão online deste conteúdo.



Podcast disponível em

REFERÊNCIAS

PERKOVIC, L. **Introdução à computação usando Python: um foco no desenvolvimento de aplicações**. Rio de Janeiro: LTC, 2016.

SEBESTA, R. W. **Conceitos de linguagens de programação**. 11. ed. São Paulo: Bookman, 2018.

EXPLORE+

Para ter desafios mais complexos e exercícios para treinar, recomendamos uma visita ao website Python Brasil.

Acesse o site das bibliotecas apresentadas e busque pela documentação para que você possa conhecer todas as funcionalidades que estão disponíveis.

CONTEUDISTA

Humberto Henriques de Arruda

 **CURRÍCULO LATTES**