

Python estruturado

Prof. Humberto Henriques de Arruda

Prof. Kleber de Aguiar

Descrição

Identificação de estruturas de controle, de decisão e de repetição na linguagem Python, bem como dos conceitos, 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 a leitura deste conteúdo, é necessário possuir uma versão de um interpretador Python e o ambiente de desenvolvimento PyCharm (ou outro ambiente que suporte o desenvolvimento na linguagem Python). Também é preciso conhecer tipos de variáveis em Python, assim como realizar a entrada e saída de dados em Python.

Objetivos

Módulo 1

Estruturas de decisão e repetição em Python

Descrever as estruturas de decisão e repetição em Python.

Conceitos de subprogramas e a sua utilização em Python

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

Bibliotecas em Python

Identificar o uso correto de recursos de bibliotecas em Python.

Eventos em Python

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

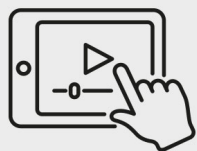
Introdução

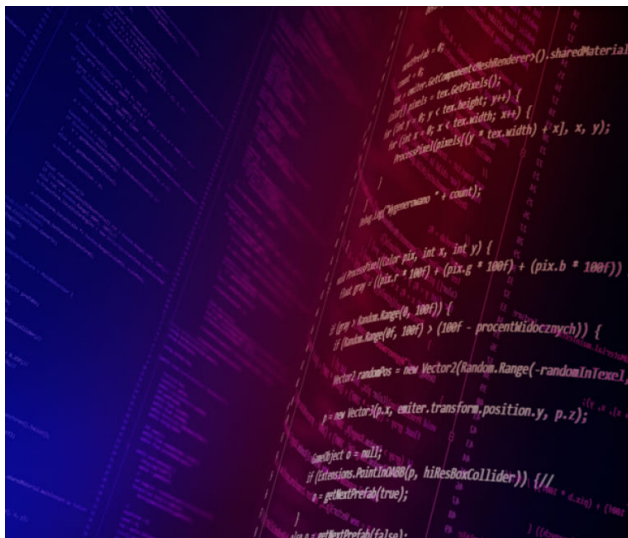
Programar significa, como em qualquer disciplina, aprender ferramentas que permitam desenvolver melhor a sua atividade. Ao conhecer os conceitos básicos de programação, o estudante desenvolve habilidades iniciais para escrever seus primeiros programas. No entanto, é difícil imaginar que aplicações profissionais sejam feitas totalmente baseadas apenas nesses conceitos básicos.

Em aplicações mais complexas, é essencial considerar a necessidade de ganhar tempo, com o computador executando as tarefas repetitivas, assim como 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 de subprogramas e bibliotecas, bem como formas de tratar exceções e eventos.

Vamos começar nossa jornada acessando os códigos-fonte originais propostos para o aprendizado de Python estruturado. Baixe o arquivo [aqui](#), descompactando-o em seu dispositivo. Com isso, você poderá utilizar os códigos como material de apoio ao longo de sua leitura!

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





1 - Estruturas de decisão e repetição em Python

Ao final deste módulo, você será capaz de descrever as estruturas de decisão e repetição em Python.

Vamos começar!



Emprego de estruturas de decisão e repetição

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



Estruturas de decisão

Tratamento das condições

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

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

Tabela: Tratamento das condições
Elaborada por Humberto Henriques de Arruda

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 as duas tabelas a seguir:

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

Tabela 2 - Estruturas de decisão simples
Elaborada por Humberto Henriques de Arruda

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	}

Estruturas de decisão compostas
Elaborada por Humberto Henriques de Arruda

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:

Python



```
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
```

A estrutura elif.

Veja uma implementação possível com a estrutura elif no emulador a seguir (trata-se do código 1 no arquivo disponibilizado na introdução deste texto). Clique em Executar:

Exercício 1

TUTORIAL

COPIAR

Python3

```
1 idade = eval(input('Informe a idade da criança: \n'))  
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 criança deve ser vacinada contra a gripe.')  
7 else:  
8     print('A criança não precisa de vacinação.')  
9
```

null

null



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

Estruturas de repetição

Estruturas de repetição for

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

Antes de detalharmos o **for**, vamos conhecer uma função de Python que gera uma lista de valores numéricos. Essa lista nos 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, desde uma maneira simples até a mais complexa. Observe:

Simples

Ela pode ser chamada de maneira simples, tendo apenas com um argumento. Nesse caso, a sequência começará em 0 e será incrementada de uma unidade até o limite do parâmetro passado (exclusive). Exemplo: `range(3)` cria a sequência (0, 1, 2).

Não iniciadas em 0

Para que a sequência não comece em 0, pode-se informar o início e o fim como parâmetros. Lembre-se de 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, na ordem, os parâmetros de início, fim e passo. O passo é o valor que será incrementado de um termo para o próximo. 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:

```
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 a seguir:

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
Instrução fora do for	}

A estrutura for

Elaborada por Humberto Henriques de Arruda

Vamos analisar um exemplo simples em Python: imprimir todos os elementos de uma sequência criada com a chamada **range()**. Observe adiante uma possível implementação desse exemplo (código 2 no arquivo disponibilizado na introdução deste conteúdo). Clique em Executar no emulador seguinte:

Exercício 2

 TUTORIAL  COPIAR

Python3

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

null

null



1. A linha 1 mostra a criação do laço com a variável item percorrendo a sequência (2, 5, 8), que é criada pela chamada range(2, 9, 3);
2. A linha 2 indica a instrução que será executada para cada repetição desse 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 seguinte.

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

Um exemplo do for em Python
Elaborada por Humberto Henriques de Arruda

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 demonstrada a seguir (no arquivo disponibilizado na introdução, trata-se do código 3).

Informe um nome (string) no campo Input do emulador e clique em Executar:

Exercício 3

TUTORIAL

COPIAR

Python3

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

null

null



1. A linha 1 faz com que a palavra inserida pelo usuário seja armazenada na variável nome;
2. A linha 2 mostra a criação do laço, com a variável letra percorrendo a sequência de caracteres armazenada na variável nome;
3. A linha 3 indica a instrução que será executada para cada repetição desse laço. O laço for executará a instrução da linha 3 tantas vezes quanto forem os elementos da sequência que está na variável nome.

Uso do laço for com qualquer sequência

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

Podemos utilizar o laço for com iterações sobre qualquer sequência - e não somente as numéricas e as strings.

Observe este exemplo (código 4 do arquivo disponibilizado):

Exercício 4

TUTORIAL COPIAR

Python3

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

null

null



Clique em Executar no emulador anterior e veja o resultado da execução do código.

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:

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

Tabela: Comparação do while em Python e em C
Elaborada por Humberto Henriques de Arruda

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

Uma possível implementação desse exemplo em Python aparece no código adiante (código 5 no arquivo disponibilizado neste texto).

Atenção!

Para que o exemplo seja executado corretamente, deve-se inserir, no campo Input do emulador, uma sequência de palavras, uma por linha, sendo que a última tem de ser a palavra “sair”.

No emulador seguinte, clique em Executar:

Exercício 5

TUTORIAL

COPIAR

Python3

```

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

null

null



1. A linha 1 representa a solicitação ao usuário para que ele insira uma palavra a ser armazenada na variável palavra;
2. A linha 2 cria o laço while, que depende da condição <valor da variável palavra ser diferente de 'sair'>;
3. 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, encerrado;
4. A linha 4 representa a impressão da mensagem fora do laço while.

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 de do-while. A diferença básica entre elas é o momento em que a condição é testada, como podemos verificar 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:

Python



```
1 while True:
2     Bloco que será repetido indefinidamente
```

Exemplo

Você deseja criar uma aplicação que permaneça por meses ou anos sendo executada, registrando a temperatura ou a umidade de um ambiente. Logicamente, você possui essa informação disponível a partir da leitura de algum sensor. Você, portanto, precisa tomar cuidado e ter a certeza de que seu uso é realmente necessário para evitar problemas de consumo excessivo de memória.

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 exemplificarmos o uso dessa instrução, vamos voltar ao primeiro exemplo do laço while, utilizando o laço infinito. O laço será encerrado quando o usuário inserir a palavra 'sair'.

Atenção!

Para que o exemplo seja executado corretamente, deve-se inserir, no campo Input do emulador, uma sequência de palavras, uma por linha, sendo que a última tem de ser a palavra "sair".

No emulador abaixo, clique em Executar (trata-se do código 6 do arquivo disponibilizado):

Python3

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

null

null

Caso haja vários laços aninhados, o **break** será relativo ao laço em que estiver inserido.

Veja o código a seguir:

Python



```
1 while True:
2     print('Você está no primeiro laço.')
3     opcao1 = input('Deseja sair dele? Digite SIM para isso. \n')
4     if opcao1 == 'SIM':
5         break # este break é do primeiro laço
6     else:
```

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. Essa instrução 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 (código 8 do arquivo disponibilizado) no emulador abaixo:

Python3

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

null

null



Para ressaltar a diferença entre as instruções **break** e **continue**, vamos alterar a linha 3 do código anterior, trocando a instrução **continue** pela instrução **break**. Clique em Executar no emulador e veja a nova execução. Essa alteração está no arquivo disponibilizado neste texto (código 9).

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 é falsa.

Suponha que queiramos imprimir somente os números ímpares entre 1 e 10. Uma implementação possível está no emulador seguinte (código 10 do arquivo disponibilizado):

Exercício 9 TUTORIAL COPIAR

Python3

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

null

null



Claramente, seria possível reescrever a condição do **if-else** para que pudéssemos transformá-lo em um **if** simples, sem **else**. Entretanto, 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.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

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

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



Visão geral

Características gerais dos subprogramas

Os subprogramas são elementos fundamentais dos programas. Por isso, eles são importantes no estudo de linguagens de programação.

Neste módulo, abordaremos os conceitos de subprogramas, como características gerais, passagem de parâmetros e recursividade. Além disso, falaremos sobre a utilização deles em Python.

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

1. Cada subprograma tem um único ponto de entrada.
2. 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.
3. Quando a execução do subprograma termina, o controle sempre retorna para o chamador.

Definições básicas

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

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

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

O subprograma estará ativo após o início de sua execução, que se dá a partir da sua chamada, e enquanto ele não for 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 (chamados de funções) traz, na ordem, o tipo de retorno, o nome e a lista de parâmetros, como pode ser verificado adiante:

```
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 e 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 a seguir (código 11 do arquivo disponibilizado). Digite 1 ou 2 no campo Input do emulador e, depois, em Executar:

Exercício 10

TUTORIAL COPIAR

Python3

```
1 escolha = input("Escolha uma opção de função: 1 ou 2\n")
2 if escolha == "1":
3     def func1(x):
4         return x + 1
5     s = func1(10)
6
```

null

null



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

Parâmetros, procedimentos e funções

Parâmetros

Usualmente, um subprograma executa cálculos e operações a partir de dados que ele deve processar. Há duas maneiras de o subprograma obter esses dados: acessar 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 fazer 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 dele e a lista de parâmetros a serem vinculados a seus **parâmetros formais**, que são denominados **parâmetros reais ou argumentos**.

No exemplo anterior, em que usamos o emulador, há 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` no próximo código (no arquivo disponibilizado, trata-se do código 12):

Exercício 11

TUTORIAL COPIAR

Python3


```

1 - def largada(distancia, multiplicador=1):
2     km_rodado = 2
3     valor = (largada + distancia *
4     km_rodado) * multiplicador
5
6

```

null

null



1. Observe que, mesmo com a definição da linha 1 de dois parâmetros formais, a chamada da função na linha 9 ocorre apenas com um parâmetro real.
2. 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 nela, o que não significa o retorno de qualquer função definida pelo usuário.

Procedimentos e funções

Os subprogramas podem ser 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 ambos, as funções podem ser definidas sem retornar qualquer valor, tendo um comportamento de procedimento. Esse é o caso de Python.

Veja o código a seguir (código 13 no arquivo disponibilizado na introdução deste texto):

Exercício 12

TUTORIAL COPIAR

Python3

```

1 - def func1(x):
2     x = 10

```

```
3     print(f'Função func1 - x = {x}')
4
5
```

null

null

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

Ambientes de referenciamento local

Variáveis locais

Quando um subprograma define as próprias variáveis, ele estabelece ambientes de referenciamento local. Essas variáveis são chamadas de **variáveis locais**, nas quais seu escopo usualmente é o corpo do subprograma.

As variáveis locais podem ser:

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 vamos explicado mais à frente.

Atenção!

Nas linguagens C e C++, as variáveis locais são dinâmicas da pilha, a menos que elas sejam especificamente declaradas como `static`. Todas as variáveis locais em Python são dinâmicas da pilha. As variáveis globais são declaradas em definições de método; além disso, qualquer variável declarada global em um método precisa ser definida fora dele. Caso haja uma atribuição à variável local com o mesmo nome de uma variável global, a global é implicitamente declarada como local.

Voltemos ao exemplo do tópico Procedimentos e funções. Vamos detalhar as funções `func1(x)` e `func2(x)`:

1. 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;

2. Analogamente, a função `func2(x)` – definida nas linhas 6, 7 e 8 – recebe o parâmetro `x` e tem uma variável de mesmo nome, mas com valor atribuído 20;
3. O programa principal tem uma variável global de mesmo nome `x`, cujo valor atribuído é 0, na linha 11;
4. 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 ser executada, cada uma dessas funções tem a própria variável local a quem todas as referências internas são feitas.
5. Mesmo com a variável global tendo valor nulo, cada variável local das funções `func1(x)` e `func2(x)` tem o próprio valor. Não há 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 no código anterior (trata-se do código 14 no arquivo disponibilizado):

Exercício 13

 TUTORIAL  COPIAR

Python3

```
1 def func1():
2     global x
3     x = 10
4     print(f'Função func1 - x = {x}')
5
6
```

null

null



Observando a execução do código anterior, 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`, alterada na execução da `func2(x)`, 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. Contudo, as linguagens C e C++ não permitem essa prática.

Veja o exemplo a seguir (código 15 no arquivo disponibilizado neste conteúdo). No campo Input do emulador, digite o valor que será atribuído à variável `dist` e clique em Executar:

Exercício 14

 TUTORIAL  COPIAR

Python3

```
1 def taximetro(distancia):
2     def calculaMult():
3         if distancia < 5:
4             return 1.2
5         else:
6             return 1.5
```

null

null



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 existentes para transmiti-los ou recebê-los dos subprogramas chamados. Os parâmetros podem ser passados principalmente por:

Valor

O parâmetro formal funciona como uma variável local do subprograma, sendo inicializado com o valor do parâmetro real. Dessa maneira, não ocorre uma 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 esse parâmetro 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 forem ponteiros utilizarão a passagem por valor.

O método de passagem de parâmetros de Python é chamado de passagem por atribuição. Como todos os valores de dados são objetos, toda variável é uma referência para um objeto.

Ao se estudar a 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.

Funções recursivas

Recursividade

Uma função recursiva é aquela que chama a si mesma. Veja o exemplo da função `regressiva()` (código 16 do arquivo disponibilizado):

Python



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

```
3 regressiva(x - 1)
```

Exemplo de função recursiva.

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

Ao chamar a `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 a `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 a `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 a do exemplo a seguir (código 17 do arquivo disponibilizado neste texto):

Python

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

Função recursiva com condição de parada.

Uma função recursiva que termina tem:

1. Um ou mais casos básicos que funcionam como condição de parada da recursão.
2. Uma ou mais chamadas recursivas que têm como parâmetros valores mais próximos do(s) caso(s) básico(s) 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:

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


Rotacione a tela. 

Além disso, ela pode ser definida recursivamente por:

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

Rotacione a tela. 


Observe agora uma implementação recursiva da função fatorial em Python (trata-se do código 18 no arquivo disponibilizado):

```
Python 
```

```
1 def fatorial(n):
2     if n == 0 or n == 1:
3         return 1
4     else:
5         return n*fatorial(n-1)
```

Função recursiva fatorial.

Vale ressaltar que a função fatorial também poderia ter sido implementada de forma não recursiva, como é mostrado a seguir (código 19 no arquivo disponibilizado):

```
\
Python 
```

```
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):
```

Função fatorial com laço for.

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; a partir do 3º termo, cada termo é a soma dos dois anteriores.

Veja uma possível implementação recursiva de função que determina o n-ésimo termo da sequência de Fibonacci (no arquivo disponibilizado, trata-se do código 20):

```
Python 
```

```
1 def fibo(n):
2     if n == 1 or n == 2:
3         return 1
```

```
4 ▾     else:
5     return fibo(n - 1) + fibo(n - 2)
```

Função recursiva para sequência de Fibonacci.

1. A linha 2 traz as condições de parada.
2. A linha 5 traz as chamadas recursivas para calcular os dois termos anteriores da sequência.

Documentação de funções – Docstrings

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.

Verifique o código (no arquivo disponibilizado, trata-se do código 21) e a sua saída:

Python

```
1 #Determina o n-ésimo termo da sequência de Fibonacci
2 ▾     if n == 1 or n == 2:
3         return 1
4 ▾     else:
5         return fibo(n - 1) + fibo(n - 2)
6
```

Docstring da função `fibo()`.

Python

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

Exibição da docstring da função `fibo()`.

1. No código “Docstring da função `fibo()`”, a linha 2 mostra a declaração da docstring.
2. A linha 8 exibe a impressão na tela da chamada `help(fibo)`. No código “Exibição da docstring da função `fibo()`”, está o resultado da execução desse programa.

Vamos praticar alguns conceitos?

Considere o seguinte trecho de um programa escrito em Python:

```
1 def func1(x):
2     x = 10
3     print(x)
4
5
6 x = 0
```

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.

E 10 10 10

[illegible]

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

```
1 def rec(n):  
2     if n < 2:
```


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.

E Na linha 3, o return deve ter a mesma indentação do if.

Parabéns! A alternativa B está correta.

[illegible]

3 - Bibliotecas em Python

Ao final deste módulo, você será capaz de identificar o uso correto de recursos de bibliotecas em Python.

Vamos começar!



Aplicando as bibliotecas Python para a resolução de problemas reais

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



Importação de funções e módulos

Biblioteca padrão Python

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, tendo o 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.

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

1. Operações matemáticas.
2. Interface gráfica com o usuário (GUI).
3. Funções matemáticas e geração de números pseudoaleatórios.

É importante lembrar os 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, há um objeto.

Dica

Para compreender mais os conceitos de classe e objeto, pesquise sobre o 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(). Clique em Executar e observe o uso do math no próximo código (trata-se do código 22 no arquivo disponibilizado):

Exercício 14

 TUTORIAL  COPIAR

Python3

```
1 import math
2
3 x = math.sqrt(5)
4 print(x)
```

null

null



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.

Módulos nativos do Python

Módulo math

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

Listaremos agora algumas das funções do módulo math:

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...)

Principais funções do módulo math
Elaborada por Humberto Henriques de Arruda

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

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 a seguir mostra algumas das principais funções disponíveis para distribuições de valores reais no módulo random.

Função	Retorno
<code>random()</code>	Número de ponto flutuante no intervalo (00,0, 1,0)
<code>uniform(a, b)</code>	Número de ponto flutuante N tal que $a \leq N \leq b$
<code>gauss(mu, sigma)</code>	Distribuição gaussiana. mu é a média e sigma é o desvio padrão.
<code>normalvariate(mu, sigma)</code>	Distribuição gaussiana. mu é a média e sigma é o desvio padrão.

Principais distribuições de valores reais
Elaborada por Humberto Henriques de Arruda

Funções para números inteiros

Veja algumas das principais funções disponíveis para inteiros no módulo random.

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

Principais funções do módulo random para inteiros
Elaborada por Humberto Henriques de Arruda

Funções para sequências

Esta tabela mostra algumas das principais funções disponíveis para sequências no módulo random:

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

Principais funções do módulo random para sequências
Elaborada por Humberto Henriques de Arruda

Saiba mais

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 a fim de 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 estas instruções:

Acesse sua conta no Google.

Depois acesse Segurança.

Acesso a app menos seguro.

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

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

Para fazer seu primeiro envio, crie um programa no seu projeto. O código a seguir mostra as importações necessárias para o envio:

Python

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

Envio de e-mail com módulo smtplib 1.

O código seguinte mostra a criação da mensagem com o corpo e seus parâmetros:

Python

```
1
2
3
4
5
6 #criação de um objeto de mensagem
```

Envio de e-mail com módulo smtplib 2.

1. A linha 7 mostra a criação de um objeto de mensagem.
2. A linha 8 exibe o corpo da mensagem em uma string.
3. As linhas de 11 a 14 devem ser preenchidas com os valores adequados para que seu programa seja executado com sucesso.
4. A linha 17 anexa o corpo da mensagem (que estava em uma string) ao objeto msg.

O próximo código mostra os passos necessários para o próprio envio:

```
Python
```

```
1
2
3
4
5
6
```

Envio de e-mail com módulo `smtplib` 3.

1. As linhas 20 e 21 mostram a criação do servidor e a sua conexão no modo TLS.
2. A linha 24 mostra o login na conta de origem do e-mail.
3. A linha 27 representa o envio propriamente dito.
4. A linha 30 exibe o encerramento do servidor.

O código referente ao programa para o envio de e-mail a partir do servidor SMTP do Gmail encontra-se no arquivo disponibilizado neste conteúdo (código 28).

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`.

Esta tabela aponta 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.

Função	Retorno
sleep(segundos)	A função suspende a execução por determinado número de segundos.

Principais funções do módulo time
Elaborada por Humberto Henriques de Arruda

O código a seguir (código 23 no arquivo disponibilizado) mostra um exemplo de chamada das funções **time()** e **ctime()**.

Exercício 15

TUTORIALCOPIAR

Python3

```
1 import time
2
3 x = time.time()
4 print(f'Local time: {time.ctime(x)}')
```

null

null



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.

Aparecem os seguintes valores :

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]

Índice	Atributo	Valores
8	tm_isdst	0,1 ou -1
N/A	tm_zone	Abreviação do nome da timezone

Principais funções do módulo time
Elaborada por Humberto Henriques de Arruda

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 a 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. O código adiante mostra a criação da sua primeira janela, ainda sem qualquer elemento gráfico.

Python



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

Primeira janela com tkinter 1.

1. 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();
2. Para exibir textos, vamos usar o elemento Label. O próximo código 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 inferior direito de coordenadas (200,200).

Python



```
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()
```

Primeira janela com tkinter 2.

Veja o resultado de sua primeira janela, apenas com o texto, na imagem a seguir.



Exibição da primeira janela com tkinter.

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

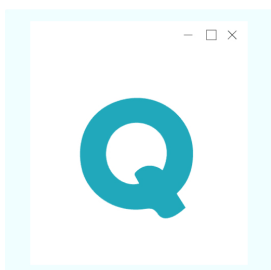
```
Python
```

```
1 from tkinter import *
2
3 def funcClicar():
4     print("Botão pressionado")
5
6 janelaPrincipal = Tk()
```

Segunda janela com tkinter.

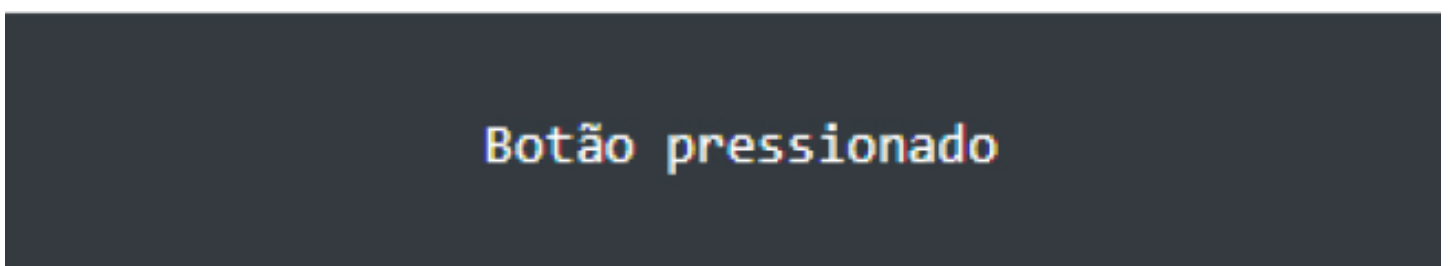
1. No código anterior, há a inserção do elemento de imagem e o botão. Nas linhas 10, 11 e 12, é feita a criação do objeto Label para conter a imagem e seu posicionamento. Observe que passamos a utilizar o método pack(), que coloca o elemento centralizado e posicionado o mais perto possível do topo, depois dos elementos posicionados anteriormente.
2. O elemento botão é criado na linha 14 com os atributos text e command, os quais são respectivamente o texto exibido no corpo do botão e a função a ser executada quando o botão é clicado.
3. Para o funcionamento correto do botão, é preciso definir a função funcClicar(), nas linhas 3 e 4. Essa função serve apenas para imprimir na tela a string "Botão pressionado".

Verifique o resultado na imagem seguinte:



Segunda janela com tkinter exibida.

Tendo implementado sua janela, clique no botão para ver o resultado no console, como mostra a imagem a seguir:



Resultado do clique no botão da segunda janela.

O código referente ao programa gráfico com o uso do módulo Tkinter encontra-se no arquivo disponibilizado neste conteúdo (código 29).

Saiba mais

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

Pacotes externos

Usando pacotes externos

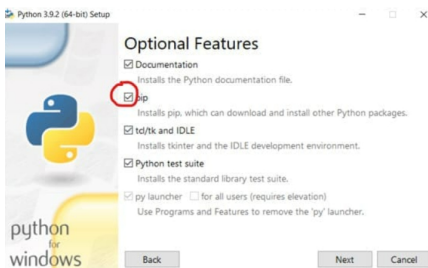
Além dos módulos fornecidos de forma integrada pela biblioteca padrão do Python, a linguagem possui uma grande vantagem: sua característica open-source permite que qualquer desenvolvedor, com o conhecimento adequado, desenvolva a própria biblioteca e os próprios módulos, os quais chamaremos, a partir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

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, 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 por meio do **pip**, um programa que pode ser instalado 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.

Atenção!

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 sem esquecer de marcar a opção de incluir o pip durante o processo de instalação.

Além do pip, é necessário ter em mãos 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:

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**”.

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).

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 apontado na imagem a seguir.



Adicionando o Python ao PATH.

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

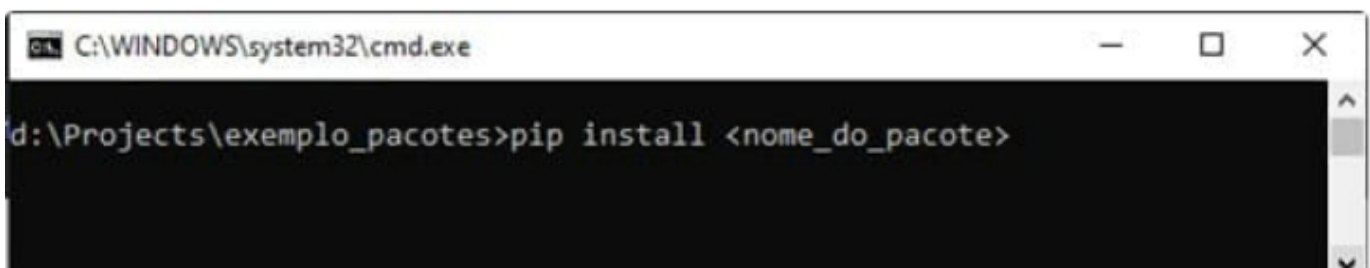
Atenção!

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.

Uma das vantagens disso é que você consegue saber exatamente quais pacotes externos estão sendo usados no projeto. É possível usar esses pacotes 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 ajudá-lo 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"**), ativar o seu ambiente virtual (se você estiver usando um) e digitar o seguinte comando: `\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 da documentação oficial do Python.

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

Nome do módulo	Pra 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

Pacotes externos mais comuns e utilizados no mercado.
Elaborada por Humberto Henriques de Arruda

Dica

Antes de começar o 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 à utilização daqueles da biblioteca padrão. Basta, para isso, utilizar o `import nome_do_modulo` no seu código.

Criação do próprio módulo

Os desenvolvedores podem criar os 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. O **arquivo do módulo precisa estar na mesma pasta do arquivo para onde ele será importado**.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 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. O código correto é:

A

```
import math
n = 5.9
print(ceil(n))
```

```
import math
n = 5.9
```

B	<pre>math.ceil(n) print(n)</pre>
---	----------------------------------

```
C import math
n = 5.9
print(ceil.math(n))
```

```
D import math
n = 5.9
print(math.ceil(n))
```

```
E import math
n = 5.9
ceil.math(n)
print(n)
```

Parabéns! A alternativa D está correta.

[illegible]

Questão 2

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

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.

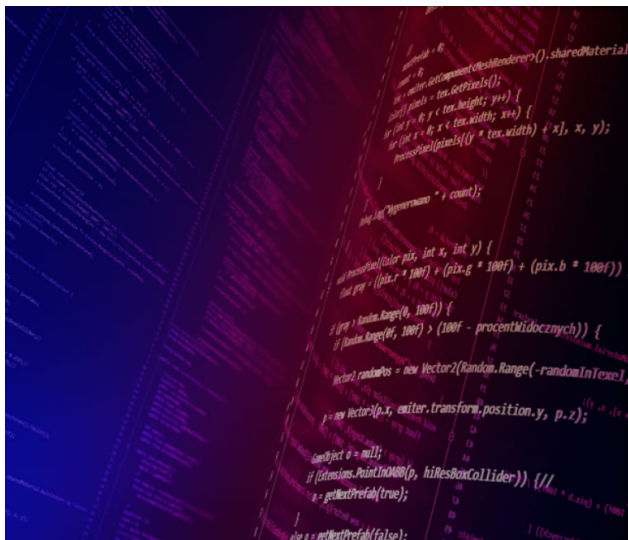
C Tem módulo de interface de e-mails **smtplib**, que não permite envio de e-mails por servidores gratuitos.

D Tem módulo de operações matemáticas **math**, que não permite operações com números complexos.

E O módulo `math` é usado para implementar geradores de números aleatórios.

Parabéns! A alternativa D está correta.

[illegible]



4 - Eventos em Python

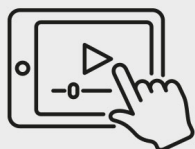
Ao final deste módulo, você será capaz de analisar as formas de tratamento de exceções e eventos em Python.

Vamos começar!



Tratamento de exceções na linguagem Python

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



Erros em um programa Python

Erros e exceções

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.

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, que é chamado analisador ou parser.

Veja exemplos nas duas imagens a seguir:

```
>>> print 'hello'

File "<input>", line 1
    print 'hello'
    ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print('hello')?
```

Erro de sintaxe 1.

```
>>> lista = [1 ; 2 ; 4]

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

Erro de sintaxe 2.

Além deles, existem os **erros que ocorrem em tempo de execução do programa**, que não se devem a uma instrução escrita errada, e sim ao fato de que o programa entrou em um estado indevido.

Elencamos os seguintes exemplos:

1. A divisão por 0.
2. A tentativa de acessar um índice indevido em uma lista.
3. Um nome de variável não atribuído.
4. Um erro causado por tipos incorretos de operando.

Em cada caso, quando o programa atinge um estado inválido, é dito que o interpretador Python **levanta uma exceção**. Isso significa que é criado um objeto que contém as informações relevantes sobre o erro.

A tabela a seguir 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.

Tipos comuns de exceção

Elaborada por Humberto Henriques de Arruda

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.

Tratamento de exceções e eventos

Captura e manipulação de exceções

Para evitar que os programas sejam interrompidos quando uma exceção é 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.

O próximo emulador mostra uma implementação simples desse exemplo:

Exemplo simples de input

TUTORIALCOPIAR

Python3

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

null

null



Mas o que aconteceria se o usuário digitasse uma palavra em vez de números? Faça essa experiência e digite uma palavra, como, por exemplo, dois, no emulador anterior e clique em Executar.

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.

Bloco try

O bloco try é executado primeiramente. Devem ser inseridas nele as instruções do fluxo normal do programa.



Bloco except

O bloco `except` só será executado se houver o levantamento de alguma exceção.

Isso permite que o fluxo de execução continue de maneira alternativa. O emulador seguinte mostra uma implementação possível desse exemplo (código 25 no arquivo disponibilizado):

Exemplo simples de input

TUTORIAL

COPIAR

Python3

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

null

null



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

Python



```
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 que executa o trabalho de captura da exceção caso não haja um tratamento explícito feito pelo desenvolvedor. É esse manipulador o responsável pela exibição das mensagens de erro no console.

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.

O emulador a seguir traz uma possível variação do exemplo anterior com a captura apenas das exceções do tipo `NameError` (código 26 do arquivo disponibilizado):

Python3

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

null

null



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.

O emulador seguinte mostra um exemplo de implementação da captura de exceções de múltiplos tipos (no arquivo disponibilizado, trata-se do código 27).

Python3

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

null

null



1. 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**.
2. 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:

```
Python
```

```
1 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 de exceções. Assim como no caso das exceções ocorridas em tempo de execução, podemos tratar os eventos criados por ações externas, como as 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.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Considere o seguinte trecho de um programa escrito em Python:

```
Python
```

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

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.

E O programa vai imprimir imediatamente o numero digitado.

Parabéns! A alternativa D está correta.

[illegible]

Questão 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 usar a cláusula catch.

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

E As cláusulas *else* e *finally* não são opcionais.

Parabéns! A alternativa B está correta.

[illegible]

Considerações finais

Neste conteúdo, você aprendeu a usar as estruturas de controle, sejam elas de decisão ou de repetição. Em seguida, foi apresentado aos conceitos de subprogramas em Python e pôde implementar as próprias funções, além de conhecer e utilizar as bibliotecas.

Por fim, você analisou as formas de tratamento de exceções e eventos. Com tal conteúdo, você certamente terá condições de desenvolver aplicações muito mais complexas e com muito mais recursos.

Abordamos os principais pontos relacionados à programação estruturada em Python e como os recursos apresentados podem auxiliar no desenvolvimento dos programas.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.



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