Modularização e uso de bibliotecas do Python

Prof. Frederico Tosta de Oliveira

Prof. Kleber de Aguiar

Descrição

Apresentação básica de como organizar os programas de forma a facilitar o reuso, trabalho em equipe e testes, por meio de utilização de funções e criação de módulos. Introdução à Interface Gráfica com o Usuário (GUI) em Python utilizando a biblioteca tkinter.

Propósito

Descrever como construir funções e módulos em Python, de maneira que o programador tenha capacidade de dividir seu problema em problemas menores, permitindo alcançar um código reutilizável, testável e com mais colaboração da equipe. Estruturar o uso de biblioteca gráfica em Python para construção de interfaces.

Preparação

Antes de iniciar o conteúdo deste tema, sugerimos que você tenha uma versão do software Python instalado em seu computador. Ele será utilizado nos módulos. Você pode pesquisar e baixar no site oficial da Python. Nos exemplos, utilizamos a IDE PyCharm.

Objetivos

Módulo 1

Organização, trabalho em equipe, reuso e testes de software

Reconhecer organização, reuso, trabalho em equipe e testes.

Módulo 2

Desenvolvimento e uso de funções em Python

Estruturar a construção e utilização de funções em Python.

Módulo 3

Módulos em Python

Formular a construção e utilização de módulos em Python.

Módulo 4

Interface Gráfica com o Usuário (GUI)

Identificar os fundamentos de Interface Gráfica com o Usuário (GUI) utilizando a biblioteca tkinter.

Introdução

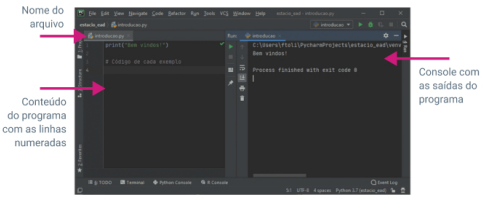
Nós, seres humanos, naturalmente, reutilizamos soluções anteriormente encontradas para resolver novos problemas. Mesmo em um novo contexto, temos capacidade para abstrair e adaptar as soluções.

No desenvolvimento de software não é diferente. De uma forma básica, a reutilização de software é o uso de softwares já existentes ou do conhecimento de um software para construir um outro novo. Isso vem sendo feito desde que se iniciou a programação, porém o reuso passou a ter aspecto distinto no estudo da engenharia de software a partir de 1968 (KRUEGER, 1992).

A proposta da reutilização é prover softwares de qualidade, aumentar a produtividade nos seus desenvolvimentos e maximizar os lucros. A motivação principal para o reuso é que cada vez mais deseja-se desenvolver sistemas maiores, mais complexos, menos caros e que sejam finalizados no prazo. Como artefatos de software reutilizáveis, podemos citar: código, especificações de projetos, requisitos e casos de testes.

Para dar suporte à reutilização do código, o Python, que é uma linguagem **orientada a objetos**, permite dividir o código em módulos, de forma que eles podem ser incluídos e utilizados por terceiros.

Portanto, ao longo do tema, utilizaremos alguns padrões que podem variar em relação à versão. A título de exemplo, veja o código a seguir em que temos, à esquerda, o arquivo contendo o código do programa e, à direita, o console contendo a saída do programa.



IDE PyCharm.

1 - Organização, trabalho em equipe, reuso e testes de software

Ao final deste módulo, você será capaz de reconhecer organização, reuso, trabalho em equipe e testes.

Organização e reuso de software

É extremamente útil organizar nosso código em partes menores quando o programa começa a crescer muito. Isso ajuda a simplificar a manutenção do código e a sua compreensão. Veja a seguir algumas vantagens (HUNT, 2019):

Simplicidade

Quebrar um grande problema em pedaços menores ajuda no desenvolvimento de soluções mais focadas, acarretando simplicidade na solução. A combinação das pequenas soluções leva à saída do problema como um todo. Se considerarmos que cada pequena solução é um módulo, teremos módulos mais simples.

Manutenibilidade

Facilitar a modificação de um módulo simples e até mesmo sua evolução, pois, por ter seus limites bem definidos, verificar se ele continua íntegro após alguma modificação é muito mais simples. A identificação de erros em códigos menores também é facilitada. Ou seja, a manutenção de módulos pequenos é mais fácil.

Facilidade para testar

Resolver um problema específico. Por isso, cada módulo pode ser testado de forma isolada, mesmo que os demais módulos do programa não estejam prontos.

Reuso

Definir módulos que realizam tarefas específicas facilita seu reuso em outros programas, visto que suas funcionalidades são bem definidas.

Apesar dos benefícios, nem sempre a reutilização está inserida no contexto organizacional e gerencial.

Dica

O primeiro passo é fazer uma avaliação do investimento e da disponibilidade da organização para adotar o reuso. Assim, escopos diferentes podem ser estabelecidos dependendo da natureza do negócio.

Os objetivos do reuso devem ser claramente definidos em função dos benefícios esperados, que podem variar entre produtividade, compatibilidade, qualidade, tempo, possibilidade de novas oportunidades etc. Iniciando-se por um projeto piloto de pequenas dimensões, é possível proceder de forma gradual, expandindo o escopo à medida que o processo do reuso amadurece.

É importante especificar o tipo de reuso que se almeja. Nesse contexto, surgem os termos: **reuso vertical** e **reuso horizontal**. Veja cada um deles a seguir:

Reuso vertical

Uma análise do domínio é iniciada para produzir uma arquitetura orientada ao reuso de desenvolvimentos de componentes, de forma

cuidadosa e planejada, que pode proporcionar altos níveis de reuso, porém pode-se levar muitos anos para alcançar esse fim.

Reuso horizontal

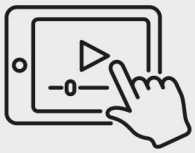
Os componentes são genéricos, mas que podem ser de utilidade para outras partes de uma organização. Os benefícios no reuso horizontal podem ser vistos em pouco tempo.

Os programas de reuso mais efetivos se concentram em identificar e desenvolver componentes pequenos e úteis e garantir que os usuários (programadores) se sintam motivados e saibam como utilizá-los.

Introdução à orientação de objetos

Neste vídeo, conheceremos o paradigma de programação chamado Programação Orientada a Objetos.

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



Desenvolvimento de sistemas em equipe

Atualmente, diversos programas facilitam o trabalho de desenvolver sistemas em equipe. Uma das principais funcionalidades disponíveis por esses programas é o controle de versão. Veja a evolução desses programas abaixo:

1986

O CVS (Concurrent Versions System) foi publicado em 1986 e permitia a múltiplos usuários acessar, modificar e atualizar arquivos sem prejudicar uns aos outros.

2000

Posteriormente, o CVS foi substituído pelo SVN (Subversion), com mais funcionalidades e menos limitações.

2005

Atualmente, um sistema de controle de código bem conhecido é o Git.

Em todos eles, basicamente, o código fica armazenado em um repositório central, em um servidor.

Veja o que acontece durante o processo de desenvolvimento:

Etapa 01

Um arquivo com código-fonte é criado em nosso computador.

Etapa 02

O arquivo é enviado para o servidor.

Etapa 03

Um código é editado em nosso computador.

Etapa 04

A modificação é enviada para o servidor.

Etapa 05

O servidor compara com a versão armazenada e tenta unificar o código.

Etapa 06

O desenvolvedor recebe uma mensagem solicitando que ele mesmo faça essa junção, caso o servidor não consiga unificar o código devido a algum conflito.

Etapa 07

O desenvolvedor, após resolver o conflito, envia novamente o arquivo para o servidor, que passa a ser a última versão válida.

Etapa 08

O código atualizado fica disponível para todos que têm acesso ao repositório.

Algumas ferramentas, como o GitLab, permitem trabalhar com integração contínua. A cada modificação no código enviada ao servidor, ele compila, testa e valida o novo código antes de efetivar a modificação no repositório. Isso visa garantir a integridade do código fonte existente.

Tipos de testes de software

Os testes são tão importantes que existem metodologias de desenvolvimento baseadas em teste, como o Test Driven Development (TDD) ou Desenvolvimento Guiado por Testes.

Dica

Durante o desenvolvimento de programas, devemos constantemente testar nosso código, de forma a certificar que não existam erros.

Nessa metodologia, as descrições dos testes são escritas antes mesmo do código, começando com o desenho e desenvolvimento dos testes para cada pequena funcionalidade do programa.

> **O objetivo do TDD é fazer com que o código fique limpo, simples e livre de erros.**

Porém, não desanime, provavelmente você já criou um teste e nem sabe. Você sabia que quando rodamos nossa aplicação e verificamos as funcionalidades que acabamos de implementar estamos realizando testes? Isso mesmo! Estamos realizando um tipo de teste chamado **teste exploratório**, que é uma das formas de realizar testes manualmente.

> **Nos testes exploratórios, não temos um plano de testes a seguir, apenas estamos, como o próprio nome já diz, explorando as funcionalidades da aplicação.**

Para ter um teste manual completo, precisamos de um plano, ou seja, uma lista de todas as funcionalidades que a aplicação tem, todos os tipos de entrada de dados e todos os resultados esperados. Para cada modificação feita no código, é necessário executar novamente todos esses testes, de forma a garantir que a aplicação continue funcionando sem erros. Cansativo, não?!

Felizmente, existem formas de automatizar os testes. A automatização dos testes requer um plano de execução de testes, que deve contar com as partes da aplicação que se deseja testar, a ordem de execução e quais resultados são esperados para cada teste.

Atenção!

Esse plano, em Python, é um **script**, que pode ser executado sempre que houver uma modificação no programa.

Os testes podem ser divididos em testes unitários (unit test) e testes de integração (integration test). Veja a seguir a diferença entre um e outro:

Testes unitários

Verificam uma pequena parte da aplicação.

Testes de integração

Verificam como os diversos componentes interagem entre si.

Ambos os testes são suportados em Python.

Atividade discursiva

Para ilustrar, vamos realizar um teste unitário para a função sum (soma) do Python. Vamos seguir o seguinte plano:

Criar uma lista com os números 5, 10 e 20;
Executar a função sum utilizando a lista criada como argumento;
Verificar se o resultado da soma é 35, que é a soma de todos os itens da lista (esperado sucesso);
Verificar se o resultado da soma é diferente de 35 (esperado erro).

Digite sua resposta aqui...

Exibir solução

Confira o código a seguir:

Python

```
>>> lista = [5, 10, 20]
>>> soma = sum(lista)
>>> assert soma == 35, "Aguardando sucesso!"
>>> assert soma !=35, "Aguardando erro!"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

Para fazer as verificações, utilizamos a palavra reservada **assert**, seguida de uma expressão que permita ser avaliada como verdadeira ou falsa (soma == 35 ou soma != 35) e uma descrição do nosso teste.

Caso o teste apresente erro, a execução do programa será suspensa e será exibida a descrição do erro, para que possamos localizá-lo.

Para facilitar a criação e execução de planos de teste mais complexos, o Python disponibiliza uma biblioteca interna chamada unittest, que dá suporte tanto ao TDD quanto a testes em geral. Essa biblioteca permite automatizar os testes, criar rotinas para iniciar serviços dos quais os testes dependem e agregar os testes em coleções.

Muitas ferramentas de gerência de repositório, como o GitLab, integram-se a ferramentas de testes, de forma que toda modificação no código enviado ao repositório desencadeie uma sequência de testes. Isso é feito para garantir integridade do código contido no repositório.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Em qual modalidade de reuso de software os componentes reutilizáveis são genéricos e os benefícios esperados da reutilização, como produtividade, qualidade etc. podem ser vistos em pouco tempo?

A Reuso vertical

B Reuso horizontal

C Reuso total

D Reuso transversal

E Reuso parcial

Parabéns! A alternativa B está correta.

O reuso horizontal visa desenvolver componentes genéricos em pouco tempo, de forma a se beneficiar desses componentes prontamente. Como exemplo, podemos citar a criação de classes mais genéricas, que possam ser utilizadas em aplicações de diferentes domínios.

Questão 2

No exemplo a seguir, temos um programa em que declaramos uma lista com alguns números. Após a declaração, percorremos os itens da lista utilizando o laço for. Para cada item da lista, utilizamos uma expressão que será avaliada pelo assert. Essa expressão verifica se um número é par. Confira o código a seguir e responda:

Python

```
>>> lista = [0, 2, 4, 6, 7, 8, 9]
>>> for numero in lista:
...         asset numero % 2 == 0, "Número ímpar encontrado! ->" + str(numero)
```

Quantas mensagens de erro "Número ímpar encontrado! ...” serão exibidas?

A Nenhuma.

B	1.
C	2.
D	3.
E	O programa apresentará outra mensagem de erro.

Parabéns! A alternativa B está correta.

O assert no Python gera uma exceção que, quando não é tratada, interrompe a execução do programa. Assim, apenas uma mensagem de erro será exibida, quando o programa atingir o primeiro número ímpar da lista, o 7.

2 - Desenvolvimento e uso de funções em Python

Ao final deste módulo, você será capaz de estruturar a construção e utilização de funções em Python.

Visão geral

Funções em Python

Conforme explicado no módulo anterior, uma das formas para facilitar o desenvolvimento colaborativo e aumentar a reutilização é a divisão de um programa em partes menores. Em Python, temos algumas formas de fazer isso. Uma delas é a criação de funções, que veremos neste módulo.

> **A função é um bloco de instruções que executa uma determinada tarefa e, uma vez definida, pode ser reutilizada em diferentes partes do programa.**

As funções permitem que o código fique mais modular, possibilitando executar a mesma tarefa diversas vezes sem a necessidade de reescrever o código. A modularização facilita a manutenção e divisão das tarefas.

Como exemplo de funções disponibilizadas pelo Python, temos o print, dir, help, entre outras.

Atividade discursiva

Para ilustrar a importância da modularização, imagine que criamos um programa que calcula o IMC (Índice de Massa Corpórea) de um usuário e retorna a sua classificação: **baixo peso**, **peso adequado**, **sobrepeso** e **obeso**.

Nesse programa, existem duas formas de entrar com os dados do usuário: pela linha de comando e pela leitura de um arquivo. Para cada uma das formas, recebemos os dados de peso e altura, calculamos o IMC e a classificação do usuário. O código inserido em dois locais diferentes no programa para calcular o IMC foi o seguinte.

Digite sua resposta aqui...

Exibir solução

Python

```python
peso, altura = (entrada)
imc = peso/altura**2

if imc < 18.5:
        print("Baixo peso")
elif imc < 25:
```

As classificações de IMC implementadas foram: abaixo de 18.5, baixo peso; maior ou igual a 18.5 e menor que 25, peso adequado; maior ou igual a 25 e menor que 30, sobrepeso; e maior ou igual a 30, obeso. O IMC é calculado pela fórmula:

$$\text{IMC} = \frac{\text{peso}}{\text{altura}^2}$$

Agora, imagine que a OMS (Organização Mundial da Saúde) resolveu alterar as faixas do IMC. Para corrigir o programa, precisamos procurar no nosso código, em dois lugares, onde está sendo realizado o cálculo e alterar cada um deles.

E se precisássemos criar um ponto de entrada de dados via interface gráfica? Precisaríamos inserir todo esse código em mais um ponto no nosso programa.

> **Não teria sido mais fácil se uma única função retornasse as faixas do IMC e depois utilizássemos essa função em diversos pontos do nosso programa? Toda correção relacionada ao cálculo do IMC seria realizada em um único trecho de código.**

O caso apresentado utiliza uma matemática muito simples, mas em algumas situações precisamos incluir nos programas cálculos mais avançados, que requerem uma dedicação maior para implementar.

Dica

A separação do código em funções facilita o gerenciamento das atividades, pois, enquanto uma pessoa desenvolve uma parte do programa, como, por exemplo, a interface gráfica, a outra pode se dedicar à implementação da função complexa.

Até agora voltamos nossa atenção basicamente ao motivo de se criar funções, mas como elas funcionam? Como executamos, ou no linguajar da computação, **chamamos** uma função dentro do nosso programa? Como é o fluxo de execução do código quando chamamos uma função?

Em um programa de computador que não utiliza técnicas de paralelismo, o fluxo de execução é linear, ou seja, no caso do Python, o seu interpretador executa uma linha por vez do nosso código.

Curiosidade

Quando, durante a execução do programa, o interpretador encontra uma função, o fluxo "salta" para executar os códigos das funções e depois retorna para continuar a execução de onde "saltou".

Observe a imagem a seguir, em que apresentamos o fluxo de execução de um programa. Os números representam a ordem de execução dos comandos e as setas o fluxo:

Fluxo de execução de um programa.

Ao iniciar o programa, apesar das funções estarem definidas no início, nenhum comando é executado, apenas informamos ao interpretador a existência delas. O interpretador começa a executar a linha 1 e ao executar linha 2 encontra uma chamada de função. Quando isso ocorre, o fluxo passa diretamente para a função, executando as linhas 3 e 4.

Após executar as instruções contidas na função, o fluxo retorna para o ponto onde foi feita sua chamada e continua a execução do programa, linhas 5, 6 e 7. Na linha 7, existe outra chamada de função. O fluxo passa para essa função, linhas 8 e 9, e retorna novamente para o fluxo principal, continuando sua execução.

Definição de funções

Como definir uma função sem parâmetros

Vamos agora ver como definir e utilizar uma função em Python. A forma mais simples de defini-la utiliza a seguinte sintaxe:

Sintaxe de uma função em Python.

Utilizamos a palavra reservada **def** seguida do nome que desejamos dar à nossa função, um par de parênteses e dois pontos.

Esse nome será utilizado para chamar nossa função em um momento posterior e, assim como nome de variável, precisa seguir algumas regras. Os nomes podem ser compostos por letras, underline (_) e números de 0 a 9, exceto o primeiro caractere.

Atenção!

Em Python, não utilizamos chaves para delimitar o corpo de uma função, como ocorre em outras linguagens de programação.

De acordo com a PEP8, basta indentar o código em 4 espaços e todas as linhas indentadas farão parte do corpo da função.

No exemplo a seguir, vamos definir uma função chamada hello que contém duas linhas. Observe que as duas linhas, que contêm o print, estão indentadas:

Definição da função hello.

Para executar, ou chamar, uma função, **escrevemos o nome dela seguido por um par de parênteses**.

Veja o exemplo a seguir no qual definimos novamente a função hello na linha 1 e chamamos essa função na linha 7:

Definição da função hello.

A sequência de execução do programa pode ser acompanhada pelas saídas no console à direita da imagem. Primeiro, é impresso **Início do Programa** (linha 6), depois **Olá** (linha 2), em seguida **Mundo** (linha 3) e, por último, **Término do Programa** (linha 8). Observe que, quando a função hello é chamada na linha 7, o fluxo muda para a linha 2, início da função, e, ao término da função, retorna para linha 8.

Como definir uma função com parâmetros

Frequentemente, as funções executam suas tarefas utilizando informações do usuário. Como nem sempre sabemos quais dados o usuário irá informar, precisamos de uma forma de passar essa informação para a função durante a execução do programa.

Curiosidade

Na computação, os dados que uma função espera receber são chamados de parâmetros.

Retorne ao exemplo apresentado na introdução deste módulo, sobre cálculo do IMC. Quais dados o usuário precisa informar? **Peso** e **altura**, certo? Esses seriam os parâmetros da função.

Para definir uma função com parâmetros em Python, fazemos como na imagem a seguir.

Definição de uma função com parâmetros em Python.

Diferente da definição anterior que tinha parênteses vazios, agora temos uma lista de nomes, ou melhor, uma lista de parâmetros. Cada parâmetro fica disponível como uma variável dentro do corpo da função. Os valores dos parâmetros são preenchidos no momento em que a função é chamada. Esses valores passados aos parâmetros são chamados de **argumentos**.

No exemplo a seguir, vamos definir a função **calcula_imc**, que contém dois parâmetros, **peso** e **altura**. A definição dessa função vai da linha 2 à linha 5, conforme vemos pela indentação. Em seguida, na linha 9, vamos chamar essa função com os argumentos **70** e **1.80**, que serão atribuídos aos parâmetros **peso** e **altura**, respectivamente.

Lembrando que **O Python obedece a ordem em que passamos os argumentos! Fique atentos!**

Definição da função calcula_imc.

Observe a saída do console e verifique que os argumentos foram passados corretamente. Dentro da função, temos acesso ao **peso** e **altura**, comprovados pelas linhas 2 e 3. Utilizamos esses dois valores para calcular o IMC (linha 4) e imprimir o resultado encontrado (linha 5).

Além de permitir chamar uma função baseando-se na ordem dos parâmetros, podemos utilizar explicitamente o nome dos parâmetros durante a chamada para identificar cada argumento.

Dica

Quando utilizamos os nomes, podemos usar os parâmetros fora de ordem, pois o Python vai utilizar esses nomes para combinar corretamente parâmetro e argumento.

Veja no exemplo a seguir, no qual, na linha 9, chamamos a função calcula_imc utilizando o nome dos argumentos fora de ordem, porém com os nomes dos argumentos explícitos. Observe pelo console que os valores foram passados corretamente e o IMC calculado continua o mesmo:

Definição da função calcula_imc.

Retorno de valores em funções

Como receber os resultados de uma função

É muito comum na programação que as funções produzam algum valor que precise ser utilizado no decorrer do programa. Nesses casos, dizemos que a função **retorna** um valor. O valor retornado, normalmente, é atribuído a alguma variável para ser armazenado e utilizado posteriormente. Para retornar um valor, ao final da função, precisamos utilizar a palavra reservada **return** seguida do valor ou variável que se deseja retornar (ou em branco, caso a função não retorne valor algum).

Relembrando

Lembre-se de que retornar um valor é diferente de imprimir um valor na tela!

Ainda no exemplo do IMC, vamos redefinir a função de forma que ela retorne do valor computado do IMC. Esse valor retornado será utilizado para verificar qual a classificação do usuário, como na imagem a seguir:

Redefinição da função calcula_imc.

Veja na linha 5 do exemplo que utilizamos a palavra chave **return** seguida da variável imc. Isso indica que a função retornará o valor contido na variável **imc**, que foi calculado na linha 4.

Em um primeiro momento, a linha 9 do programa pode parecer estranha, pois atribuímos uma função a uma variável (**indice**).

Saiba mais

Isso é muito comum e, na prática, indica que a variável indice receberá o valor retornado pela função **calcula_imc**.

A linha 10 imprime o valor retornado, que pode ser verificado no console. Nas demais linhas, o valor retornado é utilizado nas estruturas de decisão para verificar qual a classificação do IMC.

Funções de ordem superior (higher order definition)

Em Python, há funções que ao invés de retornar valores, retornam uma função. Além disso, existem funções que recebem uma outra função por parâmetro. Funções com essas características são chamadas de Funções de Ordem Superior (Higher Order Function).

No exemplo a seguir, script **selecao.py**, definimos a função superior **selecionador**, que possui dois argumentos, uma lista de inteiros **seq** e uma função **teste** (linha 2).

O argumento teste recebido é uma função que testa se um número é par ou não (**verifica_par**), retornado um valor **booelano**, **True** ou **False** (linhas 9 a 13). A função selecionador retornará uma lista contendo apenas os elementos de **seq** que forem classificados como **par**. Na parte principal do programa, função **main** (linha 16), chamamos a função selecionador passando por parâmetro uma lista de inteiros e armazenamos o seu retorno na variável números_pares (linha 18).

Por fim, percorremos a lista retornada por selecionador, imprimindo na tela os seus elementos em forma de string, "{num} -> par", sendo **num** a variável usada na iteração do laço **for** (linhas 19 a 20).

Note que se alterássemos o script **seleção.py**, substituindo a função verifica_par por uma outra função cujo retorno fosse um valor booleano, o programa seria executado sem nenhum problema:

Python

```python
# script selecao.py
def selecionador(seq, teste):
    selecionados = []
    for elemento in seq:
        if (teste(elemento)):
            selecionados.append(elemento)
```

No próximo exemplo, script **potencia.py**, implementamos a função superior **calcula_potencia**. Essa função recebe por parâmetro um número inteiro que será o expoente da potenciação (linha 2) e tem uma função interna (**potencia**), cujo argumento será a base no cálculo de potência a ser realizado por ela (linhas 3 a 5). Para esse exemplo, utilizaremos um emulador de códigos.

Na parte principal do programa, função **main** (linha 8), recebemos os valores de base e expoente informados pelo usuário do programa, por meio da função nativa do Python chamada input (linha 9). A seguir, como a função input retorna os valores em forma de string, separamos esses valores por meio da função nativa **split**, e os convertemos para inteiros, atribuindo-os às variáveis base e expoente, respectivamente (linha 11). Invocamos **calcula_potencia**, passando o valor do expoente por parâmetro, que nos retornará uma instância da função **potencia** que será então atribuída à variável **potencia_de** (linha 14). Utilizando potencia_de com o argumento base, invocamos indiretamente a função potencia para efetuar a potenciação propriamente dita, sendo que o seu retorno será armazenado na variável res_potencia (linha 15). Finalmente, imprimimos no console o resultado final (linha 16).

No campo Input do emulador abaixo, insira os valores do expoente e da base, separados por 1 espaço em branco, que serão utilizados pela função calcula_potencia. Depois, clique em Executar e confira o resultado do programa no campo Console do emulador:

Exercício 1 TUTORIAL COPIAR

Python3

```python
# script potencia.py
def calcula_potencia(expoente):
    def potencia(base):
        return base**expoente
    return potencia
```

null

null

Note que em ambos os scripts, seleção.py e potencia.py, colocamos a parte principal do programa dentro de uma função chamada **main**, que por sua vez é invocada dentro de uma estrutura de seleção **if __name__ == "__main__"**. Isso se configura em uma boa prática de programação, pois caso um desses scripts fosse importado por um outro script, a parte principal do programa (conteúdo da função **main**) não seria automaticamente executada, o que aconteceria se esses comandos estivessem "soltos" dentro dos scripts.

Comentário

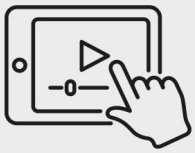
A variável especial **__name__** do Python pode ter dois valores: o nome do script ao qual ela pertence em forma de string ou a string “__main__”. Por exemplo, se um script **script_qualquer.py** é executado diretamente, o valor da sua variável **__name__** será **“__main__”**. Por outro lado, caso esse script seja importado como um módulo por um outro script, o valor da sua variável **__name__** será “script_qualquer”.

Ao se colocar a chamada a função principal de um script (convencionalmente chamada de função **main**) dentro de um **if __name__ == “__main__”**, o programador deixa garantido que a parte principal de um script só será executada quando este for executado, e não importado. Isso garante que códigos indesejados não sejam executados ao se importar um script como módulo.

Como receber os resultados de uma função

Neste vídeo, apresentaremos um resumo dos tipos de função que estudamos neste módulo.

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



Funções e escopo de variáveis

Escopo de variáveis em funções

Quando utilizamos funções, precisamos tomar cuidado ao definir e utilizar variáveis, pois toda variável tem um escopo de utilização. O escopo é o contexto de uma variável ou função durante a execução de um programa. Ele define a quais dados temos acesso em diferentes partes do programa.

> **Quando definimos uma variável fora de uma função ou laço, dizemos que a variável está no escopo global. As variáveis do escopo global mantêm os mesmos dados e estão acessíveis em todas as partes do programa, seja dentro de funções, laços ou estruturas de decisão.**

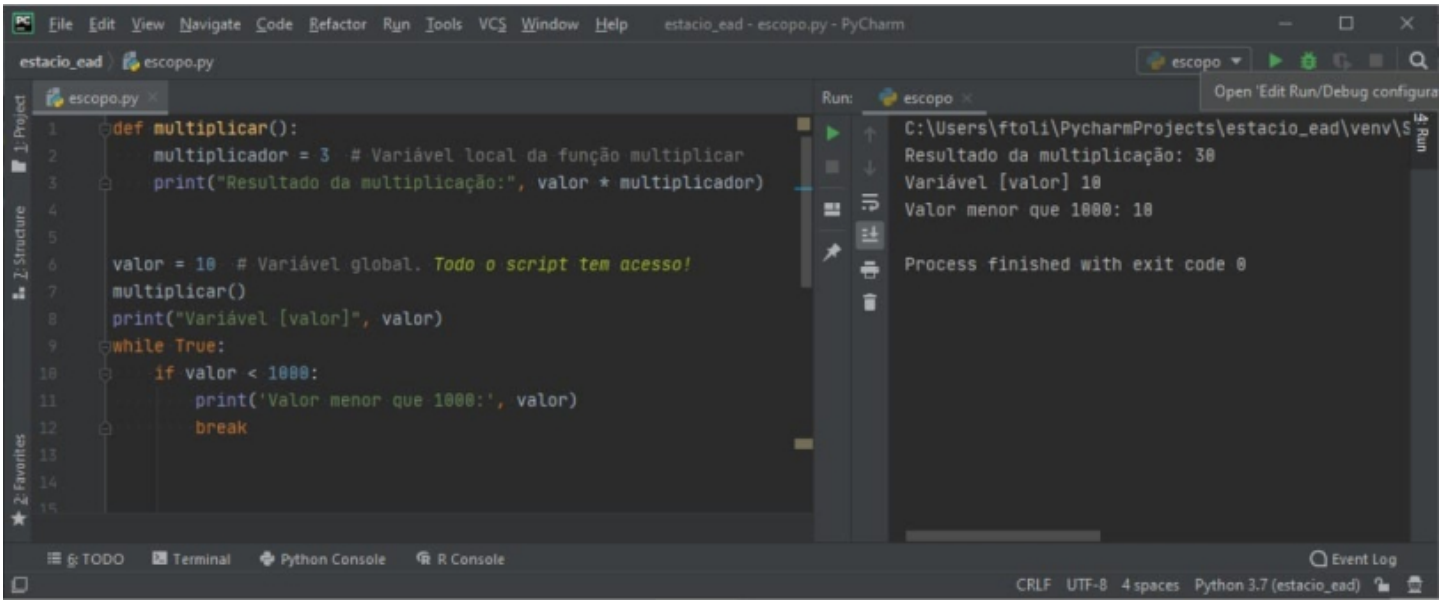
Quando declaramos uma variável dentro de uma função, dizemos que ela está no escopo local. Mas que local? O corpo da função! Ela só está disponível para ser utilizada dentro da função.

Veja no exemplo a seguir, no qual definimos duas variáveis: a variável global **valor** e a variável local **multiplicador** da função multiplicar.

Variável valor

Observe que a variável **valor** definida na linha 6 está disponível no corpo principal do script, na linha 8, dentro da função **multiplicar**, na linha 3, dentro do laço while, na linha 11, e na condição do if, na linha 10.

Ao executar o script, tudo ocorre como planejado, como podemos verificar na saída do console.

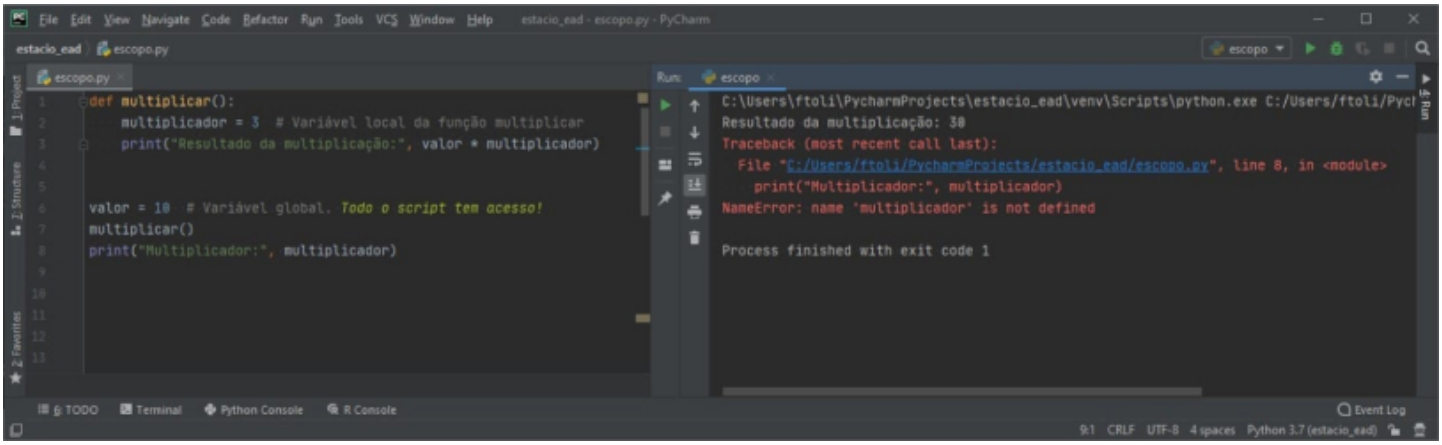


Definição da variável global valor.

Variável multiplicador

Vamos tentar acessar a variável **multiplicador** fora do corpo da função, na linha 8.

Ao executar o programa, recebemos um erro dizendo que a variável **multiplicador** não foi definida. Isso acontece, pois ela está definida apenas para o escopo local da função **multiplicar**.



Definição da variável local multiplicador.

Durante o desenvolvimento, **fique atento ao escopo da variável. Os parâmetros de uma função também são considerados variáveis locais**.

Existem casos em que a variável local recebe o mesmo nome da variável global. Como o Python resolve esse problema? O Python dá prioridade para utilização da variável local. Para forçar o Python a utilizar a variável global, precisamos utilizar a palavra reservada **global** seguida do nome da variável antes de utilizá-la. Veja no exemplo a seguir:

Utilização da variável local em Python.

Primeiro, definimos as três variáveis globais **variavel_a**, **variavel_b** e **variavel_c** nas linhas 10, 11 e 12. Dentro da função, definimos a **variavel_a** e a **variavel_b** nas linhas 3 e 4. Porém, na linha 2, comunicamos ao Python que é para utilizar a **variavel_a** global.

Assim, a linha 5 imprime o valor 1 para **variavel_a**. Na linha 6, como o Python prioriza a utilização da variável local, é impresso o valor 2. Na linha 7, como não foi definida nenhuma **variavel_c** no escopo da função, ele utiliza a sua definição global, 30. Na linha 16, imprimimos a **variavel_a** novamente. Como ela foi alterada dentro da função explicitamente por meio da palavra chave global, é impresso 1 também. Na linha 17, imprimimos o valor da **variavel_b** global, 20, e, na linha 18, o valor da **variavel_c** global, 30. Observe a saída do console da imagem anterior.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

No exemplo a seguir, temos um programa onde definimos a função teste (linha 1), que espera dois parâmetros: entrada e saída. Nas linhas 6 e 7, chamamos essa função de duas formas diferentes.

Programa da função teste.

Qual será a saída impressa pelas linhas 6 e 7 do programa?

A
carro moto
carro moto

B
carro carro
moto moto

C
carro moto
moto carro

D
moto carro
carro moto

E
moto carro
moto carro

Parabéns! A alternativa A está correta.

Quando passamos os nomes do argumento durante a chamada de uma função, o Python obedece a essa nomenclatura e atribui corretamente o valor de cada parâmetro. Caso contrário, o Python obedece à ordem com que foram passados.

Questão 2

Para o exemplo a seguir, qual o valor de c:

Função transforma.

A	15
B	20
C	25
D	35
E	30

Parabéns! A alternativa D está correta.

A função retorna o valor passado como parâmetro ao quadrado (52=25) e atribui a variável a novamente. Depois, soma as variáveis a e b (25 + 10 = 35) e atribui o valor à variável c.

3 - Módulos em Python

Ao final deste módulo, você será capaz de formular a construção e utilização de módulos em Python.

Criação e importação de módulos

Criação de um módulo

Quando escrevemos um programa de computador, não temos a dimensão que ele pode alcançar em termos de tamanho. À medida que sua dimensão aumenta, sua manutenção pode ser muito custosa, principalmente em relação ao tempo.

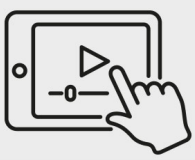
Com o intuito de minimizar o custo de manutenção como, por exemplo, a inclusão de uma nova instrução, é uma boa prática dividir o programa em arquivos menores que em geral se escreveriam como sendo um único arquivo. Além disso, outras formas de minimizar o tempo na elaboração de um programa de computador é usar instruções prontas.

O Python permite usar arquivos contendo definições e comandos. Tal procedimento é definido como módulo. O padrão do nome do arquivo é o nome do módulo acrescido do sufixo .py. No vídeo a seguir, apresentamos um breve introdução sobre o assunto.

Introdução a módulos em Python

Neste vídeo, faremos uma introdução sobre os módulos em Python.

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



Para ilustrar o processo de importação, criaremos o nosso próprio módulo e o importaremos dentro de outro script. Utilizaremos novamente o exemplo do IMC para ilustrar.

Primeiro, vamos criar um arquivo chamado **imc.py**, que será o nosso módulo de cálculo de IMC.

Recomendação

Após a criação desse arquivo, abra-o na sua IDE favorita.

Em um primeiro momento, o nosso módulo disponibilizará a função **calcula_imc**, conforme definida em exemplos anteriores.

O arquivo **imc.py** contém o código exibido na imagem a seguir:

Código do arquivo imc.py.

Como todo módulo também é um script, para verificar se tudo foi definido corretamente e não temos erro, executamos o arquivo **imc.py** e obtivemos a saída mostrada no console. Nele, temos a mensagem **Módulo IMC importado**, que é decorrência do **print** da linha 1 do módulo. Ele também será utilizado para confirmar se nosso módulo foi importado corretamente. A partir da linha 4, temos a definição da função **calcula_imc**, que retorna o valor da variável imc calculado na linha 7.

Importação de um módulo

Para utilizar esse módulo em outro script, precisamos informar ao Python que queremos importá-lo. Uma das sintaxes para importar um módulo em Python é:

Python

```
1 >>> import nome_arquivo
```

Utilizamos a palavra reservada import seguida do nome do arquivo **SEM** a extensão **py**. Para importar o módulo **imc.py**, fica assim:

Python

```
1 >>> import imc
```

Agora, vamos criar o arquivo que conterá nosso script principal, **principal.py**, e apenas importar o módulo **imc**. Para verificar se a importação foi feita corretamente, vamos executar o script **principal.py**, como mostrado na imagem a seguir:

Importação do módulo imc.

Observe a saída do console. Verifique que foi impresso **Módulo IMC importado**. Isso indica que a primeira linha do módulo **imc** foi executada com sucesso, ou seja, nosso módulo foi importado com sucesso!

Relembrando

Os dois módulos precisam estar na mesma pasta!

Definições de módulo

Utilização das definições de um módulo

No próximo exemplo, daremos um passo adiante. Mostraremos como utilizar a função **calcula_imc** definida dentro do módulo **imc**.

Para utilizar uma função definida em outro módulo, usamos a sintaxe:

```Python
>>> nome_modulo.nome_funcao(...)
```

Onde temos o **nome do módulo** importado, seguido de um **ponto** e o **nome da função**. Para utilizar a função **calcula_imc**, temos:

```Python
>>> imc.calcula_imc(x, y)
```

Onde x e y são o peso e a altura passados como argumento, como na imagem a seguir:

Importando função imc.

No exemplo da imagem, dividimos a tela em 3. À esquerda, temos o módulo **imc**; no meio, temos o script **principal**; e, à direita, o console. Observe como utilizamos a função **calcula_imc** na linha 4 do script principal destacada a seguir:

Script principal da função calcula_imc.

Mesmo sendo de outro módulo, **temos acesso ao retorno da função**.

Nesse caso, atribuímos à variável **indice**. Acompanhe no console a sequência de execução do arquivo **principal.py** e veja que os valores continuam corretos.

O Python também permite que o programador importe apenas a função desejada, sem a necessidade de importar o módulo todo. A sintaxe para isso é:

```python
>>> from nome_modulo import nome_funcao
```

Assim, podemos utilizar a função diretamente, como se ela estivesse definida dentro do nosso script. Dessa maneira, não **teremos acesso ao nome_modulo**, mas apenas à função importada.

> **Caso tenha alguma função importada com o mesmo nome de uma função do script, a função do script tem prioridade.**

Observe o exemplo a seguir, em que, na linha 1, importamos a função **calcula_imc** do módulo **imc** e a usamos diretamente na linha 4:

Importação da função calcula_imc.

Veja pelo console que os resultados continuam os mesmos.

Para finalizar o nosso módulo **imc**, criaremos mais uma função, chamada **classifica_imc**. Essa função retorna uma **string** com a classificação do IMC a partir de um IMC informado. Essa função contém apenas um parâmetro, o valor do IMC.

Veja como ficou a t no módulo **imc**. Observe que estamos retornando à classificação do IMC, não imprimindo, como estávamos nos exemplos anteriores:

Definição da função classifica_imc.

Apesar de as duas funções estarem no mesmo módulo, **elas são independentes**. **Podemos chamar uma ou a outra**.

Dica

Para utilizar essa função no nosso script principal, precisamos importá-la também.

Veja na linha 2 a importação da função e na linha 7 como ela foi chamada. Pegamos o valor retornado da função **calcula_imc** e armazenamos na variável **indice**, que foi utilizada como argumento para função classifica_imc. Por sua vez, o valor retornado pela função **classifica_imc** foi armazenado na variável **classificacao_imc**, para posteriormente, na linha 8, ser utilizado para imprimir o resultado, como podemos observar a seguir:

Resultado módulo imc.

Com isso, finalizamos o nosso módulo **imc**.

É possível importar módulos atribuindo a eles um nome reduzido (**alias**). Isso é muito útil quando se trata de módulos com nomes longos. Vamos a um exemplo usando o nosso módulo **imc**:

Python

```python
import imc as c
from imc import classifica_imc as c_imc

índice = c.calcula_imc(altura=1.88, peso=70)
classificação_imc = c_imc(índice)
```

Assim como importamos o nosso modulo **imc**, podemos importar outros módulos do Python como:

Math

Disponibiliza diversas funções matemáticas.

Os

Disponibiliza funções de sistema operacional, entre outras.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Durante o desenvolvimento de um script, vimos a necessidade de criar um módulo separado. Esse módulo se chama **meu_modulo**. No script principal, **meu_programa**, importamos o módulo e começamos o desenvolvimento, conforme imagem a seguir:

Importação do módulo meu_modulo.

Qual a saída do programa ao executar o script **meu_programa**?

A
Meu novo Script
Meu novo Módulo

B
Meu novo Módulo

C
Meu novo Módulo
Meu novo Script

D
O programa vai apresentar um erro.

E
Meu novo Script

Parabéns! A alternativa C está correta.

Ao importar um módulo, executamos todo seu conteúdo. Como a primeira linha do meu_programa é a importação do módulo, executamos o conteúdo do módulo meu_modulo e obtemos Meu novo Módulo. Na linha 4, imprimimos Meu novo Script.

Questão 2

Considere os scripts a seguir e responda:

Importação da função triplica.

Qual a saída impressa ao executar o script meu_programa?

A 15 15 15

B 15 45 135

C 5 5 5

D O programa vai apresentar um erro.

E 5 15 45

Parabéns! A alternativa D está correta.

Como importamos apenas a função triplica, não temos acesso ao módulo meu_modulo como escrito na linha 6.

4 - Interface Gráfica com o Usuário (GUI)

Ao final deste módulo, você será capaz de identificar os fundamentos de Interface Gráfica com o Usuário (GUI) utilizando a biblioteca tkinter.

Biblioteca Tkinter

Utilização da biblioteca Tkinter

Dentre vários ambientes de desenvolvimento do Python, existem interfaces gráficas do usuário (GUI – Graphic User Interface). Estas tornam a interação com o usuário muito mais fácil e dinâmica.

Um dos ambientes gráficos que permitem desenvolver interfaces em Python é a Tkinter.

A biblioteca Tkinter utiliza o conceito de orientação a objetos para implementar seus elementos gráficos. Como ela não faz parte do núcleo do Python, precisamos importá-la, assim como fizemos com o nosso módulo imc. No video a seguir, falaremos mais sobre as interfaces gráficas com o usuário.

Interface gráfica

Neste vídeo, conheceremos mais sobre as interfaces gráficas com o usuário.

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

Para os exemplos que seguem, criamos um arquivo chamado **interface.py** para conter o script da nossa interface.

Inicialmente, importamos o módulo tkinter na linha 1 e chamamos a função **_teste()** do módulo tkinter na linha 3. Essa função abre uma janela para verificar se está tudo instalado corretamente. Observe a imagem a seguir:

Importação do módulo tkinter.

Ao executar o script **interface.py**, obtivemos a seguinte janela, indicando que tudo está correto:

Janela do script interface.py.

Para criar nossa própria interface, precisamos criar a janela principal que conterá todos os nossos componentes, botões, texto, menu, listas etc.

Para criar essa janela, utilizamos a classe **Tk** do módulo tkinter para instanciar esse objeto e atribuir a uma variável, como a seguir:

```Python
>>> principal = tkinter.Tk()
```

A sintaxe para criar um objeto a partir de uma classe é igual à chamada de uma função, porém, em vez de recebermos apenas um valor como retorno, recebemos um objeto, que contém atributos e métodos (funções). Nesse caso, a variável **principal** é um **objeto** do tipo **Tk**.

Essa variável será utilizada como referência para os outros componentes da interface, indicando que ela é a janela principal do programa.

Dica

Como última linha do script, precisamos chamar o método mainloop() do objeto **principal**.

O método nada mais é do que uma função interna do objeto que deve ser chamado usando a sintaxe:

```Python
>>> nome_objeto.nome_metodo()
```

Todo o código da interface deve ficar entre a definição da variável e o **mainloop()**, como na imagem a seguir:

Posição do código da interface.

Ao executar esse programa, obtemos a seguinte janela em branco:

Janela em branco.

Outros exemplos de métodos do objeto do tipo Tk são: **winfo_height()**, que retorna a altura da janela, e **title()**, que retorna o texto apresentado no título da janela.

No exemplo a seguir, vamos adicionar um botão à nossa interface. Primeiramente, precisamos importar a classe que implementa o botão no tkinter. Essa classe se chama Button e sua importação está na linha 2 do exemplo.

Para criar o botão, utilizamos a sintaxe exibida na linha 6, cujo primeiro parâmetro é a tela onde o botão será colocado e o parâmetro text é o texto que aparece no botão. A classe Button retorna um objeto, que foi atribuído à variável **botao**. Na linha 7, utilizamos o método place do objeto **botao** para definir as coordenadas da posição do botão na tela, como visto a seguir:

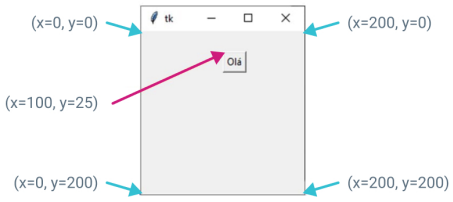
Adição de um botão à interface.

Observe como ficou nossa tela após a execução do script. Destacamos na imagem seguinte como são calculadas as coordenadas no tkinter. A tela padrão do tkinter tem 200x200 pixels. O canto superior esquerdo da tela tem coordenadas (x=0, y=0), o superior direito (x=200, y=0), o inferior esquerdo (x=0, y=200) e o inferior esquerdo (x=200, y=200).

Atenção!

O ponto de partida dos componentes é o canto superior esquerdo.

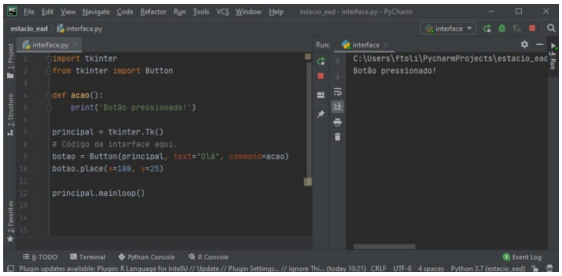
No nosso caso, o canto superior esquerdo do nosso botão está nas coordenadas (x=100, y=25). Observe a seguir:



Cálculo das coordenadas no tkinter.

Para definir a ação para esse botão, precisamos passar **uma função como argumento para o parâmetro command da classe Button**.

Observe o exemplo a seguir, em que criamos uma função chamada **acao** na linha 4, que apenas imprime no console **Botão pressionado!**. Na linha 9, passamos essa função como argumento para a classe Button.



Função acao.

A tela do programa ficou igual à anterior, porém, ao clicar no botão, é impresso **Botão pressionado!** no console.

Organização dos componentes na tela

Para adicionar uma entrada de dados, podemos utilizar a classe Entry, que representa uma caixa de texto de uma linha.

Atenção!

Normalmente, uma caixa de texto vem acompanhada de uma etiqueta (label) para indicar o que aquela caixa de texto espera como entrada: idade, nome, peso etc. No tkinter, a classe Label implementa as etiquetas.

Para adicionarmos esses dois componentes, precisamos importar cada um deles, como nas linhas 3 e 4. Na linha 14, utilizamos a classe Entry para criar a caixa de texto e atribuímos o valor retornado à variável texto. Na linha 17, utilizamos a classe Label para criar uma etiqueta. O texto da etiqueta, Label, foi passado como argumento para o parâmetro text.

> **Para organizar os elementos na tela, utilizamos o método grid que está presente nos componentes gráficos do tkinter. Esse método permite dispor os elementos como em uma tabela, utilizando como parâmetros coordenadas de linha (row) e coluna (column).**

Para ilustrar, observe a tabela a seguir, em que temos 2 linhas e 3 colunas, com as respectivas coordenadas:

row=0, column=0	row=0, column=1	row=0, column=2
row=1, column=0	row=1, column=1	row=2, column=1

Tabela: Coordenadas dos elementos na tela.
Captura de tela do PyCharm.

Como até agora só temos 3 elementos na nossa interface, eles foram dispostos apenas na primeira linha. Veja o código como ficou:

Códigos dos elementos na tela.

Pelas linhas 12, 15 e 18, temos os nossos componentes dispostos da seguinte forma:

	column 0	column 1	column 2
row 0	etiqueta	texto	botao

Tabela: Coordenadas dos elementos na tela.
Captura de tela do PyCharm.

Ao executar o programa, temos a seguinte interface:

Interface dos elementos na tela.

Alertas e entrada de dados

Entrada de dados e alertas

Outro componente que o tkinter disponibiliza é o **alerta**. Os alertas servem para mostrar alguma mensagem importante ao usuário. Normalmente, aparece na frente das outras janelas.

Atenção!

No tkinter, o módulo **messagebox** implementa alguns tipos de alerta, como alerta de erro, informação, confirmação de ação, entre outros.

No próximo exemplo, vamos utilizar um alerta para mostrar o texto inserido na caixa de texto. O alerta será exibido quando pressionado o botão **Olá**. Confira o código:

Código do componente alerta.

Na linha 5, importamos o módulo **messagebox** e, na linha 9, utilizamos a função **showinfo** para abrir o alerta.

> **O primeiro parâmetro dessa função é o título da caixa do alerta e o segundo parâmetro é o texto que estará contido na caixa. Para obter o valor digitado pelo usuário na caixa de texto, utilizamos o método get() da classe Entry, disponível na variável texto.**

Para isso, também na linha 9, utilizamos a sintaxe **texto.get()**, em que texto é o nome da nossa variável definida na linha 16.

Ao executar o programa, escrever **Olá Mundo** na caixa de texto e pressionar o botão **Olá**, obtemos a seguinte interface:

Interface de Olá mundo.

Para finalizar este módulo, vamos implementar uma interface para o nosso programa de cálculo de IMC. A nossa interface terá uma imagem, duas caixas de texto, uma para altura e outra para o peso, duas etiquetas, uma para cada caixa, e um botão. Ao apertar o botão, chamaremos as funções disponíveis no nosso módulo **imc** e utilizaremos o resultado retornado para exibir um alerta com a classificação do IMC.

A disposição dos nossos elementos deve ficar conforme tabela a seguir. Observe que a **imagem** se estende por duas linhas, chamamos isso de row span (analogamente, estender por colunas se chama column span).

	column 0	column 1	column 2
row 0	imagem	etiqueta_altura	altura
row 1		etiqueta_peso	peso
row 2			botão

Tabela: Disposição dos elementos.
Captura de tela do PyCharm

Confira como ficou o código:

Código do programa de cálculo de IMC.

O Python permite que as **importações de definições de um mesmo pacote fiquem na mesma linha, separadas por vírgula**.

Observe que, na linha 2, agora, temos todas as importações do tkinter. Na linha 3, adicionamos a importação do módulo **imc**.

Nas linhas 15 a 18, criamos uma etiqueta e, ao invés de um texto, passamos uma imagem como parâmetro.

> **Utilizamos a classe PhotoImage do tkinter e o arquivo logo.gif como argumento. O arquivo logo.gif está na mesma pasta do arquivo do nosso script (interface.py).**

A imagem está posicionada na coordenada **row=0, column=0** e, em vez de ocupar apenas uma célula, ela vai ocupar duas linhas **(rowspan=2)**.

Nas linhas 21 e 22, criamos a etiqueta Altura e a posicionamos nas coordenadas **row=0, column=1**. Nas linhas 24 e 25, criamos a caixa de texto que conterá o dado de altura entrado pelo usuário e a posicionamos à direita da sua etiqueta, **row=0, column=2**. Nas linhas 28 a 32, criamos a etiqueta e a caixa de texto para o peso, e as colocamos abaixo dos componentes da altura, **row=1**. Nas linhas 35 e 36, criamos o botão e o posicionamos abaixo de todos os componentes.

Alteramos também a ação do botão. Na linha 7, pegamos o texto inserido nas caixas de entrada peso **(peso.get())** e altura **(altura.get())** e passamos como argumento para a função **calcula_imc**.

> **O resultado retornado é armazenado na variável indice, que é utilizada para obter a classificação do imc na linha 8.**

O texto da classificação é, então, exibido no alerta criado na linha 9.

Na linha 6 do módulo **imc**, fizemos uma pequena alteração.

Dica

Como os dados entrados pelo usuário são interpretados como string, forçamos o Python a calculá-lo como float, para permitir realizar a conta.

Veja como ficou a interface final do programa:

Interface final.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Considere o código a seguir e marque a alternativa:

Biblioteca tkinter.

A — O botão de OK estará abaixo da palavra Etiqueta.

B — O botão de OK estará à direita da palavra Etiqueta.

C O botão de OK estará à esquerda da palavra Etiqueta.

D O botão de OK estará acima da palavra Etiqueta.

E O botão de OK não estará visível.

Parabéns! A alternativa B está correta.

Os elementos estão dispostos em apenas uma linha, com a etiqueta em primeiro, a entrada em segundo e o botão em terceiro lugar.

Questão 2

Baseado na interface a seguir, como poderia ser definido o grid da etiqueta 4, dado que a variável se chama etiqueta_4:

Interface da grid.

A etiqueta_4.grid(row=0, column=3)

B etiqueta_4.grid(row=3, column=1)

C etiqueta_4.grid(row=3, column=0)

D etiqueta_4.grid(row=1, column=3)

E etiqueta_4.grid(row=1, column=1)

Parabéns! A alternativa C está correta.

A etiqueta_4 está na quarta linha (índice 3) e primeira coluna (índice 0), ou seja, row -> 3 e column -> 0.

Considerações finais

A organização do nosso código é essencial durante toda a nossa vida de desenvolvedor, mesmo para os projetos em que trabalhamos sozinhos.

Em alguns casos, vamos precisar alterar um programa que fizemos há alguns anos e dificilmente lembraremos exatamente onde mexer e qual o impacto da mudança. Um plano de testes bem feito pode nos ajudar muito nessas horas!

A separação do código em funcionalidades, sejam funções, classes ou até módulos, pode nos poupar tempo em novos projetos, permitindo a reutilização dessas funcionalidades prontamente, visto que já foram testadas.

A criação de repositórios também permite um rápido compartilhamento do código-fonte, além de mostrar um histórico de alterações que fizemos durante o desenvolvimento. Em situações em que não temos um plano de testes, os sistemas de versionamento podem nos ajudar a encontrar qual modificação levou a um erro no programa, por exemplo.

Os fabricantes dos principais sistemas operacionais, Google, Apple e Microsoft, assim como iniciativas open source, Gnome Project, disponibilizam guias (guidelines) para criação de interface com o usuário para suas plataformas. A Apple apresenta guias tanto para desktop quanto para mobile. É muito importante verificar esses guias, pois essas empresas contam com o feedback de milhões de usuários!

Ouça este podcast sobre os principais assuntos abordados no tema.

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

Referências

APPLE. **Guia para interface com o usuário**. Consultado na Internet em: 20 jul. 2020.

GNOME PROJECT. **Guia para interface com o usuário**. Consultado na Internet em: 20 jul. 2020.

GOOGLE. **Guia para interface com o usuário do Material Design**. Consultado na Internet em: 20 jul. 2020.

HUNT, J. **A Beginners Guide to Python 3 Programming**. New York: Springer, 2019.

KRUEGER, C. **Software reuse**. ACM Computing Surveys 24, p.131-183, 1992.

MICROSOFT. **Microsoft Fluent 2020**. Consultado na Internet em: 20 jul. 2020.

PYTHON. **Python Software Foundation**. Consultadona Internet em: 10 jun. 2020.

TCL/TK. **Tcl Developer Xchange**. Consultado na Internet em: 10 jun. 2020.

Explore +

Além da biblioteca Tk/Tcl, é possível utilizar a biblioteca gráfica QT em Python.

O QT é uma biblioteca extremamente otimizada e permite criar aplicações para muitas plataformas, como TVs, celulares, computadores etc.

Você sabia que também é possível desenvolver aplicações WEB em Python? Confira os frameworks flask e django.

Confira o site da GitLab.