Experimentos com PYTHON Para Técnicos em ELETRÔNICA

João Alexandre da Silveira

Experimentos com PYTHON para Técnicos em ELETRÔNICA

Parte I

Python é uma linguagem de programação de computadores. Por que um técnico ou um engenheiro de eletrônica deveria conhecer essa linguagem?

João Alexandre Silveira*

Introdução

Bom, a resposta mais direta à essa pergunta é: porque Python é fácil de aprender. E também porque ela não custa nada.

Se o sistema operacional do seu PC é Linux ou MacOS, o Python já está instalado. Se for Windows você pode instalar o interpretador Python facilmente seguindo apenas alguns passos.

Mas, também, você pode usar plataformas *online* para programar em Python sem instalar nada no seu PC. Todos os recursos para a sua iniciação já estão disponíveis em qualquer PC conectado à internet.

E, por fim, porque a Eletrônica hoje se faz com *hardware* e *software* integrados. Com Python ficou mais fácil controlar 'coisas' através de uma porta digital USB de qualquer PC.

A linguagem Python tem uma *sintaxe* simples e descomplicada, mas poderosa. Sintaxe são as regras de uma linguagem. Com Python, tanto podemos criar *scripts* básicos para acionar diretamente LEDs e pequenos motores, quanto sistemas mais complexos, como a simulação em um computador de partes de uma rede especializada de neurônios biológicos; criar uma rede neural artificial e treiná-la para reconhecer padrões em grandes volumes de dados.

Scripts são textos, são roteiros, com uma série de instruções escritas para serem seguidas por pessoas em peças teatrais, cinema e programas de TV; são também *scripts* os roteiros que devem ser seguidos pelos processadores dentro dos computadores.

***Autor do livro "Experimentos com o Arduino"**

O termo vem da palavra inglesa *manuscript*, que significa 'manuscrito', ou aquilo que é escrito à mão. Nas artes cênicas, roteiristas são as pessoas responsáveis pela escrita do *script*. Nas ciências da informação quem escreve os *scripts* são os programadores; são uma série de instruções para que a máquina execute determinadas tarefas segundo uma sequência programada.

Tudo o que vamos precisar para começar nossos primeiros experimentos com a linguagem Python é de um PC *desktop* ou *notebook* conectados à internet.

Mais adiante, sim, podemos ir conectando ao nosso computador circuitos externos bem simples e controlá-los com nossos *scripts* escritos em Python. A melhor forma de aprender uma língua estrangeira é praticá-la sempre que possível. Isso vale também para qualquer linguagem de computador.

O que é a linguagem Python?

Circuitos eletrônicos dedicados, como um temporizador de segundos com o clássico CI 555, já são 'programados' desde a montagem para executarem uma mesma tarefa durante toda a sua vida útil, como ativar um outro circuito depois de contar os segundos até um certo valor.

Os circuitos modernos são mais flexíveis, no sentido de poderem ser reprogramados para executar uma outra tarefa. Esses circuitos, melhor seria chamá-los de módulos, além dos tradicionais componentes eletrônicos passivos e ativos, como resistores, capacitores e transistores, têm também como núcleo um microcontrolador, um minúsculo computador em um único *chip*. E, como qualquer computador, esses módulos precisam ser programados em uma linguagem própria. E é aqui que entramos com nossa linguagem Python.

Python é o novo BASIC? Quem se lembra da linguagem BASIC dos anos 1980 quando os primeiros computadores pessoais surgiram? Era uma linguagem de uso geral que foi criada por professores de uma universidade americana em 1964 para ensinar programação de computadores aos seus alunos.

Tanto Python quanto BASIC foram inicialmente criadas com fins didáticos e para uso geral; e estão na categoria de linguagens interpretadas. Um *interpretador* é um programa que sabe ler *scripts* escritos por humanos. O interpretador lê uma linha do *script*, verifica se a sintaxe está correta e transforma a informação dessa linha para uma linguagem de *bits* e *bytes*, a chamada *linguagem de máquina*, para que o processador finalmente execute a ordem ali codificada; só então lê a próxima linha do *script* para repetir todo o processo até a última linha.

Linguagem de máquina, também chamada de código de máquina, é a linguagem que falam os computadores digitais. É uma linguagem própria de baixo nível composta somente de dígitos binários (0's e 1's), os únicos dígitos que os processadores digitais dos computadores reconhecem. A letra maiúscula 'A' dos humanos é '0100 0001' para as máquinas.

Conforme a arquitetura adotada, cada processador digital tem seu próprio código de máquina.

Existe dois tipos de programas que convertem códigos fontes em códigos de máquinas: *interpretadores* e *compiladores*. Já vimos que os interpretadores checam e executam uma linha de cada vez de um *script*. Os compiladores leem todas as linhas do *script* de uma só vez e, se não houver erros na compilação, criam um arquivo executável, também chamado de *código objeto*, em linguagem de máquina. Uma vez o programa compilado, ele pode ser executado tantas vezes quanto preciso sem a necessidade do compilador novamente.

O Python é uma linguagem de alto nível (próxima da linguagem humana) relativamente nova, sua versão 1.0 foi lançada em janeiro de 1994. Sua versão mais nova é a 3.10.0 de outubro de 2021. Parece que seu nome foi tirado de um famoso programa da TV inglesa, 'Monty Python's Flying Circus' e atualmente é a linguagem mais popular e mais usada no mundo, até pelas gigantes da tecnologia como *Google*, *Facebook*, *Netflix*, *Amazon* e *IBM*.

Primeiros passos com Python

Um PC conectado à internet será nossa bancada de ensaios com Python. E a primeira ferramenta que vamos precisar é um editor de códigos Python, mais conhecido como *Shell*.

É como um editor de textos super simples. É um console interativo, onde podemos testar pequenos *scripts* escritos em Python. O *Shell* é uma ferramenta de linha de comando que inicializa o interpretador Python. Só não podemos salvar no PC os códigos escritos no *Shell*.

Se o sistema operacional do seu computador for *Linux* ou *MacOS* você já tem todas as ferramentas do interpretador Python instaladas.

Se seu sistema for *Windows* você tem a opção de 'degustar' antes a linguagem num interpretador Python *online* e instalar depois de testá-lo. Existe muitos interpretadores *online* disponíveis na internet, aqui vão algumas sugestões interessantes:

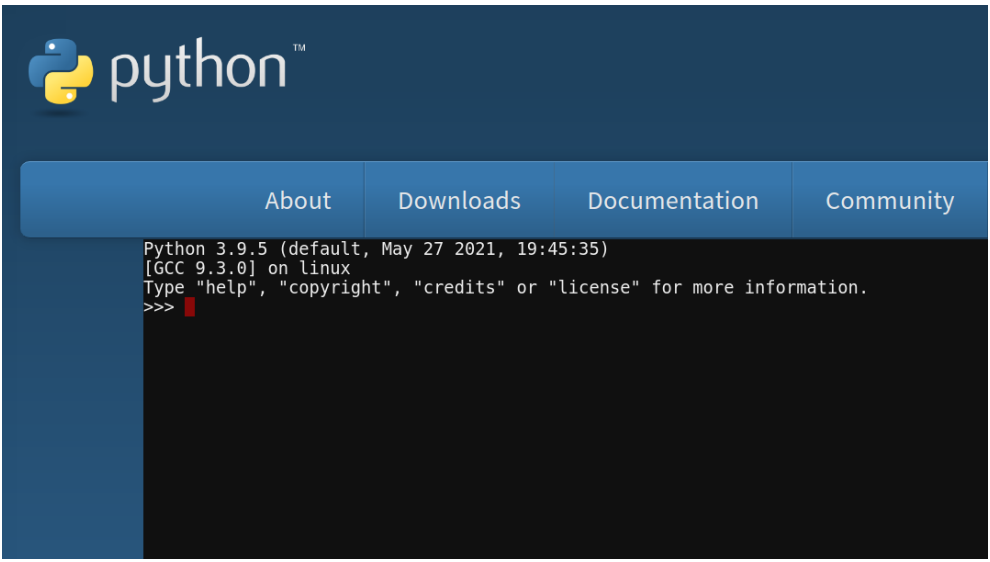
https://www.python.org/shell/

https://www.online-python.com/

https://pythoninterpreter.com/

https://www.onlinegdb.com/online_python_interpreter

Esta é a tela do interpretador Python online https://www.python.org/shell/:

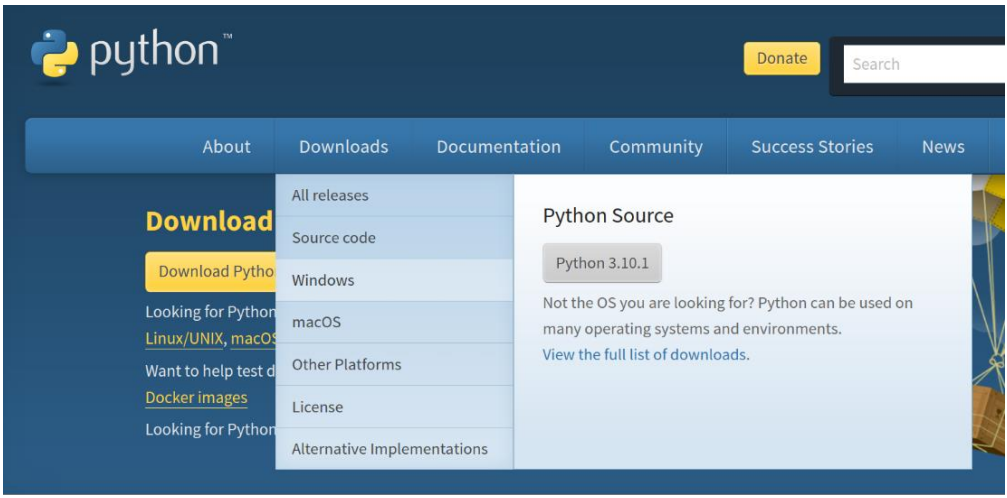


Com muito mais recursos, existe também, *online*, gratuito e sem necessidade de instalação no PC, a plataforma *Google Colaboratory* (https://colab.research.google.com/notebooks/welcome.ipynb?hl=pt_BR). Para acessá-la você vai precisar abrir uma conta no *Google*. Fácil e rápido com o seu endereço de *e-mail*.

Mais tarde, quando você já estiver bem familiarizado com a linguagem, uma outra opção aos editores *online* é a instalação no seu PC de uma ferramenta mais completa, um *IDE* (*Integrated Development Environment*), ou para nós: *Ambiente de Desenvolvimento Integrado*.

Trata-se de um editor próprio de códigos fonte Python com recursos não só de edição, mas depuração (*debug*) e execução de *scripts* em Python. Existe muitos *IDEs* gratuitos na internet para qualquer linguagem de programação de computadores.

A instalação da linguagem Python no *Windows* é igual a instalação de quaisquer outros programas nesse sistema operacional. Navegue até a página oficial da linguagem: https://www.python.org/downloads/ e na opção 'Windows' clique no botão à direita. Escolha um local para baixar o arquivo compactado e instale o núcleo da linguagem e suas ferramentas seguindo as etapas do instalador do *Windows*. (Veja a tela abaixo).



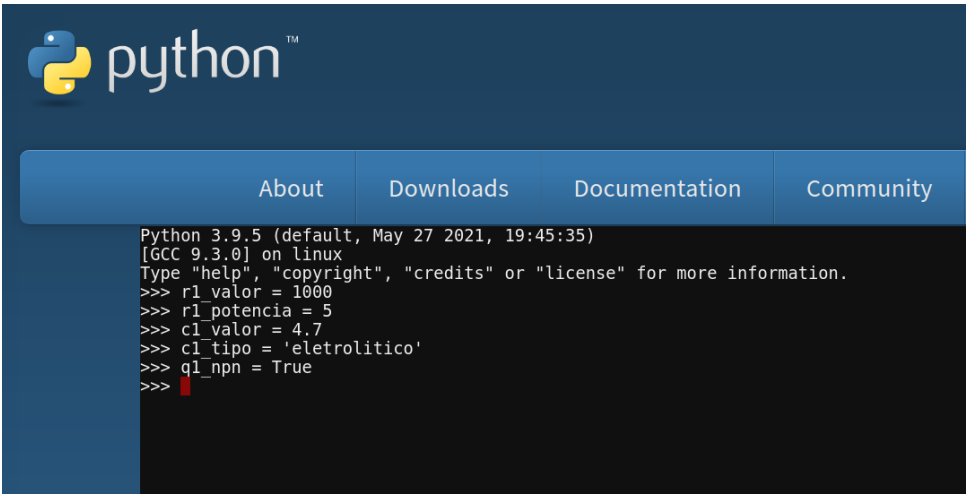
Primeiros Experimentos com Python

Antes de iniciarmos nossos experimentos, vamos entender primeiro o que são *variáveis* na linguagem Python.

Variáveis guardam valores, são como *containers* identificados. No Python tudo o que você precisa fazer para criar uma variável é inventar um nome para a variável e ter um valor para guardar nela. A variável é declarada com seu tipo no momento em que lhe atribuímos um valor. Vamos testar esse conceito.

Abra o *Shell* do Python na tela do seu PC, ou acesse o *Shell* online https://www.python.org/shell/, e espere o interpretador carregar. Quando surgir na tela 3 sinais de maior (">>>") o interpretador estará pronto para receber suas ordens. Vamos criar alguns variáveis com Python e lhes atribuir valores. Veja a tela abaixo as variáveis que criamos no *Shell online*.

Variáveis em Python são definidas com o operador de atribuição '=' (sinal de igual). Criamos uma variável em Python primeiro digitando seu nome, depois o operador de atribuição e por último o valor ou expressão que queremos guardar nessa variável.

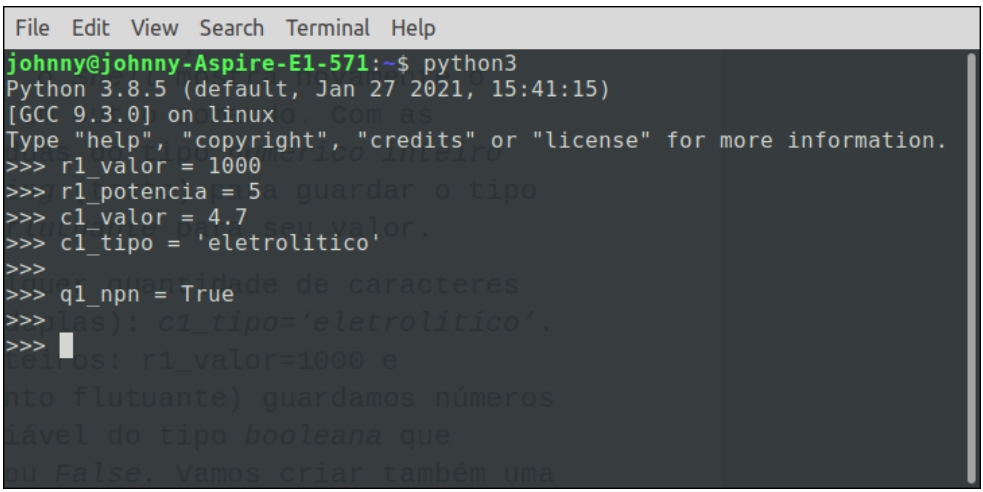


A primeira linha na tela do *Shell* acima mostra a versão e data da linguagem Python instalada.

Depois a versão do compilador *GCC* (*GNU Compiler Collection*), que converte cada linha do *script* para binários executáveis, e a plataforma em que este está sendo executado.

Na terceira linha, como conseguir ajuda sobre tópicos da linguagem, direitos autorais e outras informações. Por fim o triplo sinal de maior: '>>>' e um cursor retangular.

No *Shell* da instalação Python do Linux (tela mostrada abaixo) o cursor fica piscando.



No Shell, criamos dois containers para o resistor R1 de um circuito qualquer, os identificamos como ‘r1_valor’ e ‘r1_potencia’ e neles guardamos o valores inteiros 1000 (ohms) e 5 (watts ou V x I, ainda duas variáveis desconhecidas). Também criamos as variáveis: *c1_valor = 4.7* (*microfarads*) e *c1_tipo = ‘eletrolitico’* (sem acento) para um capacitor C1.

Observe que a cada *Enter* no teclado do PC, após conversão para código objeto, o *Shell* mostra novamente o triplo sinal de maior esperando um nova entrada, um outro comando.

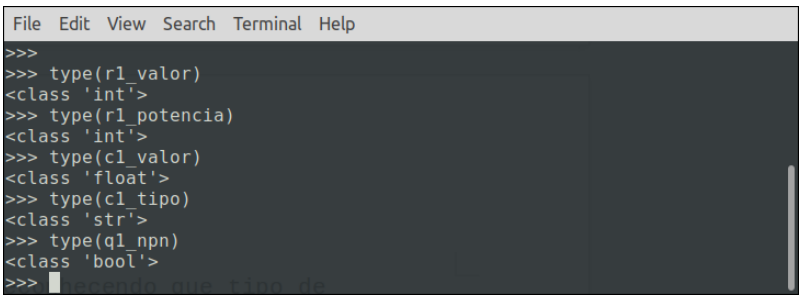
Com as entradas acima criamos três tipos de variáveis: duas do tipo *numérico inteiro* para guardar o valor e a potência de R1; uma *string* (texto), para guardar o tipo do capacitor C1, e outra do tipo *numérico ponto flutuante,* para seu valor.

Num *container* do tipo *string* podemos guardar qualquer quantidade de caracteres alfanuméricos colocados entre aspas (simples ou duplas): *c1_tipo=‘eletrolitico’*. Num do tipo inteiro guardamos somente números inteiros: r1_valor=1000 e r1_potencia=5. E numa variável do tipo *float* (ponto flutuante) guardamos números fracionários: *c1_valor = 4.7*. Existe ainda a variável do tipo *booleana* que guarda somente um de dois valores lógicos: *True* ou *False*. Vamos criar também uma variável *booleana*: *q1_npn = True*.

Os nomes das variáveis podem ser compostos por combinações de letras, números e também com o caractere ‘_’ (*underscore*); mas devem sempre começar com uma letra ou *underscore*. Para saber o valor que foi atribuído a uma variável basta digitar na tela do *Shell* o nome da variável e pressionar a tecla *Enter*.

Repare que ao criar cada variável diferente não precisamos declarar antecipadamente seu tipo, o Python já faz isso reconhecendo que tipo de informação queremos guardar.

Vamos ver se isso é verdade? O Python tem uma função interna que nos diz qual o tipo de uma variável: a função *type()*. Limpe a tela do *Shell* teclando *‘Cntl+L’* e verifique o tipo de cada variável que criamos anteriormente, passando o nome da variável como parâmetro para a função *type()*, como na tela abaixo.

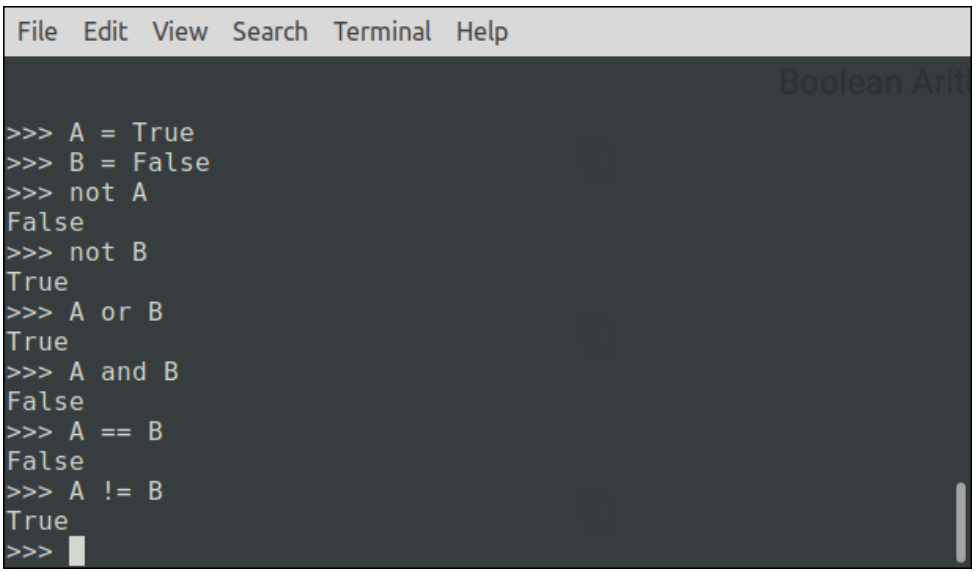


A função *type()* mostra o tipo da variável baseada no tipo de dado que ela armazena.

Vamos fazer mais um teste para entender melhor o que é uma variável booleana.

Digite no *Shell* a seguinte sentença matemática: *c1_valor > r1_valor*, e tecle *Enter*. O Python responde na tela do *Shell* que tal sentença é falsa, porque o valor numérico de C1 não é maior que o valor numérico de R1.

Mais uns experimentos com variáveis booleanas, agora com as nossos conhecidos operadores lógicos NOT, AND e OR. Reproduza no seu *Shell* as operações mostradas na seguinte tela:



'A' e 'B' são duas variáveis booleanas, uma é verdadeira e a outra falsa. O operador NOT é equivalente a uma circuito lógico inversor. Se sua entrada for verdadeira ('1'), sua saída será falsa ('0'); e vice-versa.

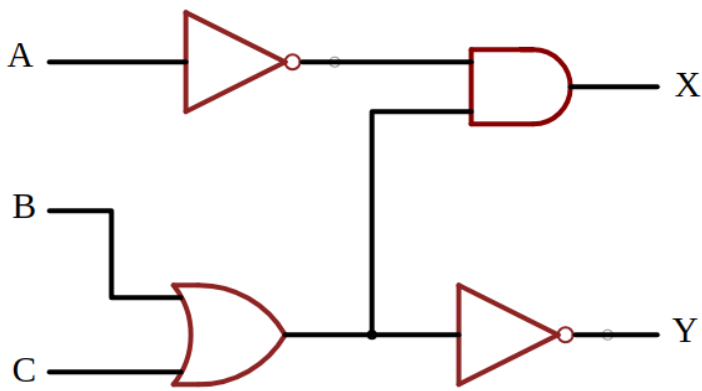
No circuito lógico OR basta uma das entradas ser verdadeira para que sua saída também seja. No AND a saída só será verdadeira se todas as entradas também forem verdadeiras.

O operador de duplo sinal de igual ('==') questiona se 'A' é equivalente a 'B'; e ao contrário, o operador '!=' questiona se essas variáveis são diferentes. Crie mais uma variável booleana, C = False, e teste as seguintes operações lógicas no *Shell* do Python: *A or (C and B) e A or (C and B).*

Funções em Python

Funções em linguagens de programação de computadores são pequenos blocos de programas com nomes próprios dentro do programa principal e que são, eventualmente, chamados para executar uma mesma tarefa.

Uma função pode ter uma ou mais entradas e uma ou várias saídas, como o circuito com portas lógicas mostrado abaixo. As entradas '*A'*, '*B'* e '*C'* seriam os *parâmetros* da função, os nomes que damos às variáveis que receberão valores, estes chamados de *argumentos*; as saídas '*X'* e '*Y'* seriam os *retornos* da função.



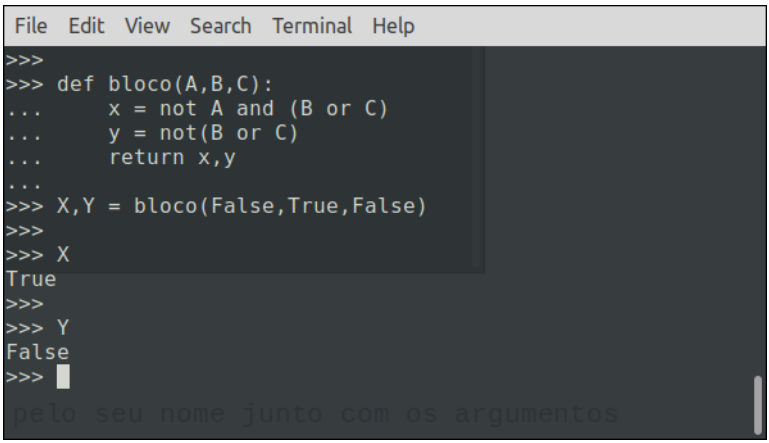
No Python temos 3 tipos de funções: as chamadas *built-ins functions*, como a função *type()* visto acima, que já vêm com a linguagem, prontas para uso imediato; aquelas declaradas (criadas) pelo programador, as *User-Defined Functions (UDFs)*; e as *funções anônimas*, chamadas de *funções lambda*, também criadas pelo programador mas que não precisam ser declaradas quando de sua criação.

As funções *UDFs* são criadas pelo programador escrevendo a palavra-chave *def()* antes do nome escolhido para a função e os nomes dos parâmetros que receberão os argumentos dentro dos parênteses, e no final da linha dois pontos.

Após teclar *Enter*, pressione também a tecla '*Tab*' para avançar 4 espaços adiante. Dá-se nome a isso de *endentação*. Agora, entre com as operações que a função deverá executar; a cada mudança de linha com *Enter* tecle *Tab* novamente.

Na última linha da função depois da palavra-chave *return* vão as variáveis que a função deve retornar. Por fim, Tecle *Enter* mais uma vez para concluir a criação da função.

OK, vamos criar uma função no *Shell* do interpretador Python que represente o circuito com portas lógicas que desenhamos acima. Veja na tela abaixo: criamos uma função chamada 'bloco' e, dentro dela, duas variáveis locais, 'x' e 'y'; cada uma vai guardar o resultado de uma operação lógica. A última linha dentro da função vai retornar a quem a chamou os valores dessas variáveis locais:



Dentro de um programa uma função é chamada pelo seu nome junto com os argumentos que vão entre parênteses.

Observe na tela acima que chamamos a função 'bloco', passamos a ela os argumentos *False,True e False* (0,1,0) e atribuímos às duas variáveis globais que criamos ('X' e 'Y') os valores das duas variáveis locais que a função vai retornar.

Veja que o retorno da função está de acordo com nosso circuito com portas lógicas: a variável 'X' guarda o valor *True* ('1') e a variável 'Y' o valor *False* ('0').

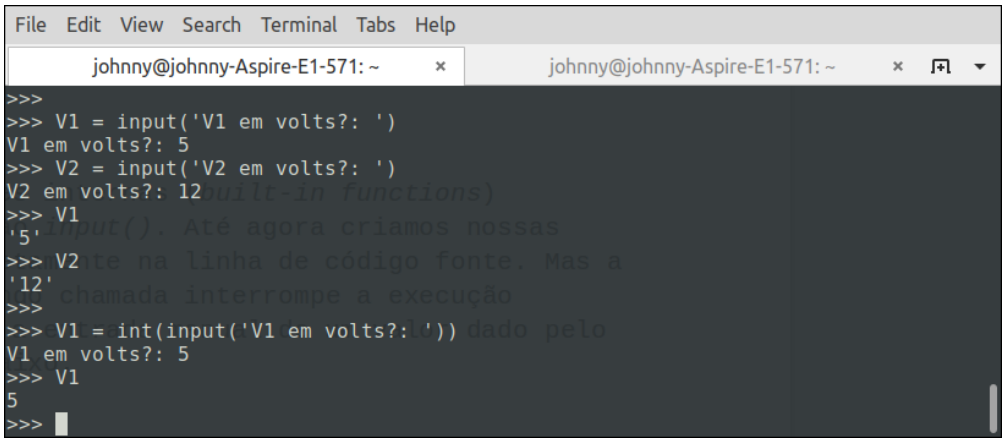
Teste o leitor outros argumentos para a função 'bloco'. Só para fechar, vamos criar uma função em Python que nos forneça a corrente que circula por R1 se temos como argumentos *r1_valor* e *r1_potencia*. Sabemos que a corrente elétrica é igual a raiz quadrada da relação entre potência e resistência, então podemos montar uma função que calcula essa corrente da seguinte forma com Python:

Aqui chamamos a função interna do Python *pow()* para exponenciar valores. Essa função aceita dois argumentos: o primeiro é a *base*, o valor que queremos elevar a uma potência; o segundo, o expoente.

A base aqui é a relação P/R (potência/resistência) e o expoente é 1/2, uma forma de calcularmos a raiz quadrada da base. Criamos uma variável *r1_imax* para esse valor da corrente em R1. Veja que também chamamos a função interna *round()* para arredondarmos o resultado da operação anterior para 4 casas decimais.

Input e Output com Python

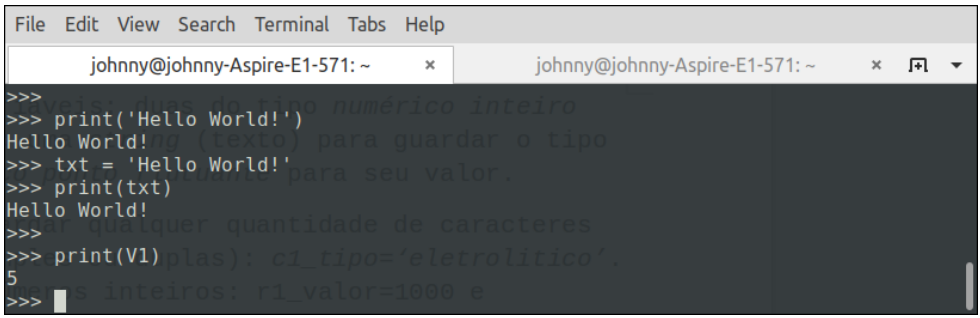
Vamos agora conhecer mais algumas funções internas (*built-in functions*) interessantes do Python. Vejamos a função *input()*. Até agora, criamos nossas variáveis atribuindo valores a elas diretamente na linha de comando, mas a linguagem Python tem uma função que quando chamada interrompe a execução sequencial do programa e fica esperando a entrada manual de um valor dado pelo usuário, a função *input()*. Veja a tela abaixo.



Aqui, atribuímos a duas variáveis V1 e V2 as entradas de dois valores pelo usuário. Mas veja, quando checamos os valores que entramos, percebemos que eles são do tipo *string* (variáveis texto, vêm sempre entre aspas). É que o *default* de saída dessa função é na forma de *string*. Então, numa linha mais embaixo, convertemos o retorno da função *input()* para número inteiro com a função *int()*.

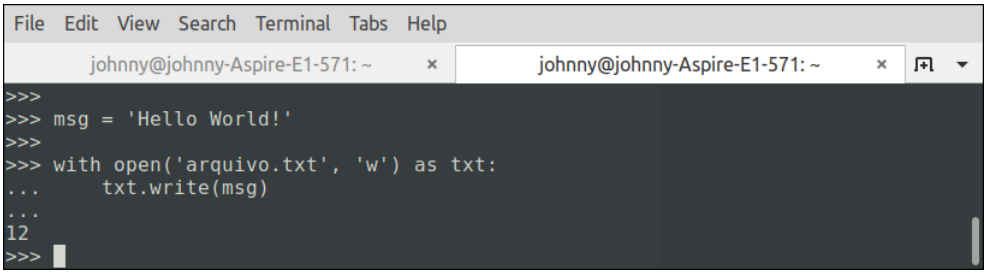
Talvez a função mais utilizada em *scripts* e programas mais complexos com Python seja a função *print()*. Equivalente ao *output* de um circuito, essa função, como seu nome sugere, mostra na tela do PC o argumento que lhe é passado entre parênteses, ou o valor de uma variável. Você já deve ter visto em algum lugar algo como: *print('Hello World!')*.

Vamos experimentar no *Shell* do Python:



Como já dissemos, todos os *scripts* editados no *Shell* do Python serão perdidos quando fechamos o interpretador ou desligamos o PC. Mais adiante, em nossos próximos trabalhos nessa série, vamos aprender como salvar e tornar executáveis nossos *scripts*.

Mas, por ora, para fecharmos essa longa introdução à linguagem Python, vamos antecipar aqui como gravar uma variável que guarda uma *string* num arquivo '.txt' no disco rígido do PC.



O número '12' que surgiu na tela corresponde ao número de caracteres do texto que foi gravado no diretório raiz do disco. Encontre o arquivo e abra-o com um editor qualquer de texto, como o *Notepad* e similares, e veja seu conteúdo.

Esperamos que esse nosso primeiro trabalho dessa série sobre a linguagem Python para técnicos em Eletrônica tenha despertado no leitor o interesse por essa fácil e elegante linguagem.

Até breve!

João Alexandre Silveira*

1- Relembrando a parte I

Seja bem vindo à segunda parte de nossa jornada pelos caminhos da linguagem Python! Antes, precisamos relembrar alguns conceitos muito importantes da linguagem, que vimos na primeira parte, publicada na revista *Antenna* de dezembro de 2021. Lá, dissemos que Python é muito fácil de aprender e que sua instalação no PC (com *Windows*) não custava nada; e que nos sistemas *Linux* e *MacOS* o núcleo do Python já vem instalado com as ferramentas básicas para se iniciar imediatamente.

Vimos que também podemos criar nossos *scripts* em Python diretamente em plataformas *online*, sem instalar nenhum programa; basta termos um PC conectado a *internet.* Tentamos convencer o leitor de que com Python ficou mais fácil controlar 'coisas' com módulos eletrônicos externos através de uma porta digital USB de qualquer PC; e dissemos que *scripts* em qualquer linguagem interpretada como o Python são roteiros com instruções escritas por humanos para o processador *(CPU)* de uma máquina 'interpretar' e executar linha por linha tais instruções, segundo sua própria linguagem, o *código de máquina* - a linguagem que falam os computadores digitais. Até comparamos o Python com a velha linguagem BASIC, quanto a terem sido criadas para uso geral e com fins didáticos por professores universitários.

***Autor do livro "Experimentos com o Arduino"**

Também falamos das *variáveis* em Python como *containers* que guardam valores. Criamos alguns desses *containers* inventando nomes que os identifiquem e neles guardamos alguns valores. O *tipo da variável* vai ser do mesmo tipo do valor que nela guardamos. Testamos esse conceito com esses *scripts* bem simples no *Shell*, o console interativo que inicializa o interpretador Python e que vem com a instalação da linguagem:

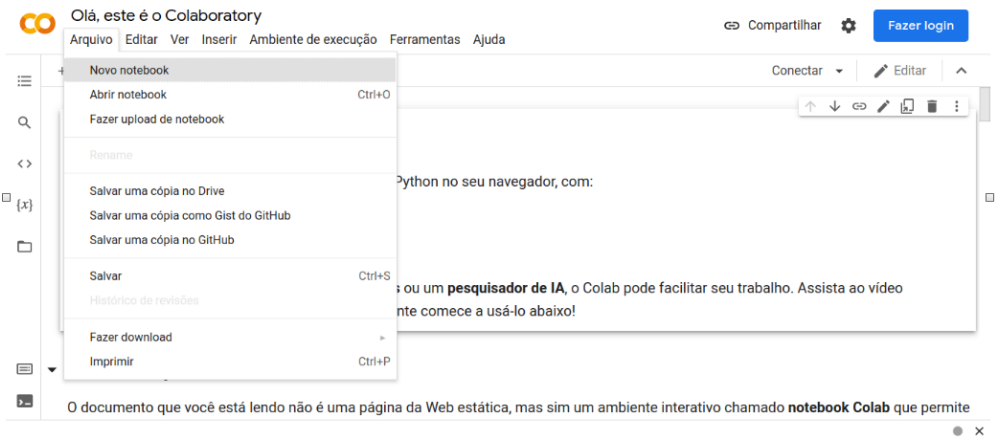
```
r1_valor = 1000     # número inteiro (integer)
c1_valor = 4.7   # ponto flutuante (float)
q1_valor = 'NPN'    # tipo string (texto)
V1_5volts = True    # booleano (True/False)
```

2- Conhecendo o editor de *scripts* Google Colab

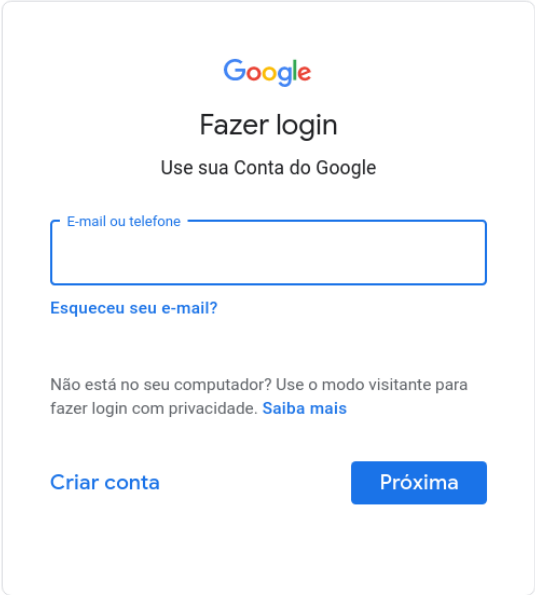
Nesta segunda parte de nosso trabalho vamos utilizar a plataforma *online* ***Colaboratory*** da *Google* como editor de *scripts* em vez do *Shell* interativo do interpretador Python. Os códigos fontes escritos no *Shell* não podem ser salvos no PC. Aqueles escritos no ***Colab***, forma como é conhecida essa plataforma, podem ser guardados numa nuvem, uma área de memória de alguns *gigabytes* em algum servidor da *Google* reservada para você quando da abertura de sua conta.

Para começar, acesse a página oficial da plataforma: https://colab.research.google.com/?utm_source=scs-index

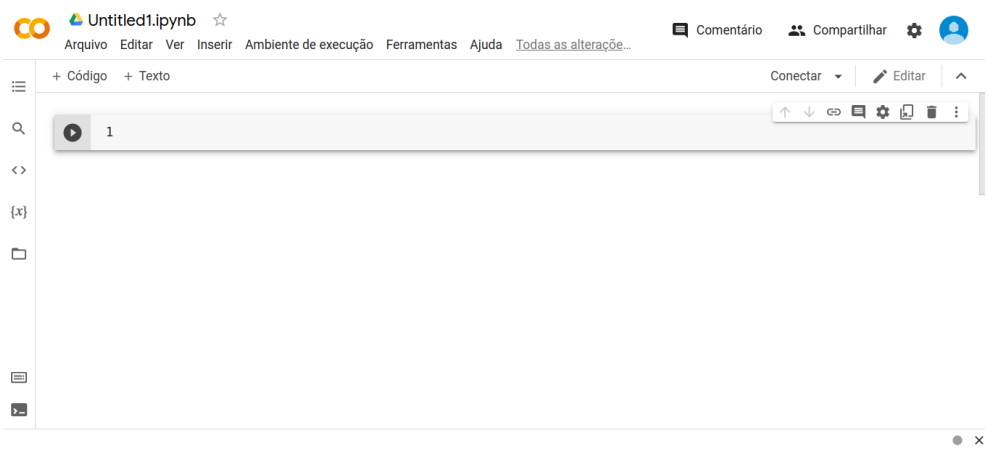
Nessa primeira página você pode fechar a apresentação do *Colab* que se sobrepõe a tela principal clicando no 'X' da janela. Depois clique em '*Arquivo'* e em '*Novo Notebook'*. Os *notebooks* do *Colab* são notebooks do *Jupyter* hospedados no *Colab*. O projeto Jupyter e o Colab são interfaces gráficas que permitem a edição de *scripts* de linguagens de computadores em qualquer navegador web.



Agora, deverá aparecer uma tela informando que para continuar é necessário fazer *login* no *Google*, ou seja, você tem que ter ou abrir uma conta no *Google Drive*. Clique no botão azul '*Fazer login'* no alto da página e entre com seu *e-mail* e *senha* se já tiver uma conta, ou crie uma clicando em *'Criar conta'*.

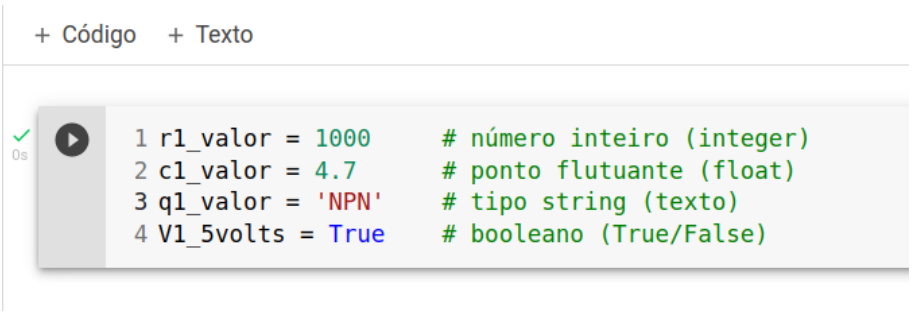


Após seu login, agora sim, a tela principal do *Colab* deverá abrir como uma nova página web, como na imagem abaixo, com um cursor piscando na primeira linha do editor de *scripts*. Se quiser, você pode fechar a primeira página aberta com a apresentação do *Colab*.



Nessa plataforma toda linha do *script* é numerada, como era na linguagem BASIC, para facilitar a depuração de erros de sintaxe e outros da linguagem, como veremos mais adiante. Experimente teclar *'Enter'* e veja como novas linhas são adicionadas às já existentes. As opções nos menus acima e os ícones à esquerda da plataforma serão mostradas conforme formos evoluindo na criação de *scripts*. Por ora, somente precisamos saber que toda vez que clicamos em *'+ Código'* um novo grupo de linhas será incluído na tela para entrada de códigos Python. À direita da tela temos alguns ícones para inserir um *link para uma página web*, abrir as *configurações do editor*, *apagar um bloco de linhas* e outras opções.

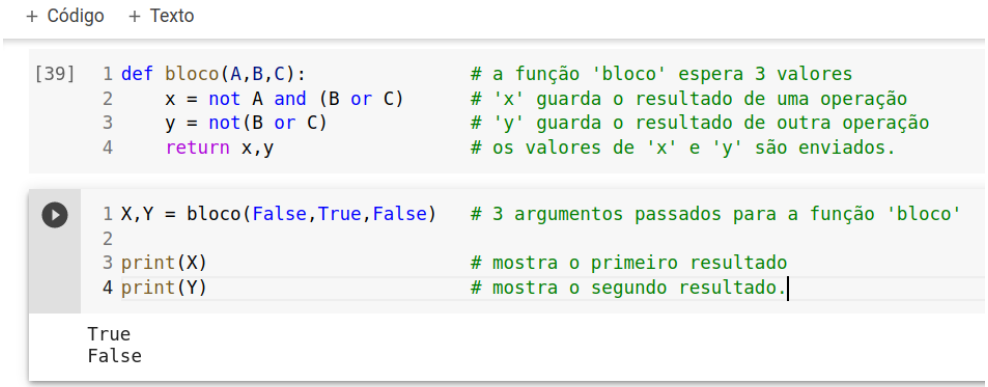
Tudo pronto? Vamos começar testando no ambiente *Colab* alguns *scripts* vistos na primeira parte dessa série:



Em cada linha, entre com um nome para uma variável e lhe atribua um valor; na mesma linha, à direita, comente essa ação em um texto curto, logo depois do símbolo *cerquilha (hash: '#')*. No Python, qualquer caractere que venha depois desse símbolo é ignorado pelo interpretador. Para executar esse bloco de comandos com quatro linhas, clique no botão de execução, à esquerda, no círculo preto com um triângulo no meio. Um sinal de *checked* na cor verde deve aparecer à esquerda do círculo preto indicando que não ocorreram erros na execução. Clique em *'+ Código'* e uma nova linha numerada surgirá, pronta para receber outros comandos em Python.

Podemos experimentar também a função que criamos para o circuito montado com portas lógicas AND, OR e NOT na primeira parte dessa série, testando-o como uma função Python, como mostrado na tela abaixo.

Logo depois, no Colab, chamamos essa função e passamos três argumentos (*False, True, False*) como parâmetros (*A,B* e *C*) para a função, que vão ser usados nas duas operações lógicas e cujos resultados serão atribuídos a duas variáveis internas: 'x' e 'y', nas duas linhas seguintes. As duas variáveis globais 'X' e 'Y' vão receber os valores das duas variáveis locais 'x' e 'y' que a função retorna.



3- Operadores no Python

Operadores, em qualquer linguagem de programação, são os símbolos que usamos para realizar (óbvio!) *operaçõe*s, sejam estas matemáticas ou lógicas. Os *operandos* são os valores envolvidos na operação que vai gerar um resultado. Por exemplo, na *operação* matemática 2+3=5, os *operandos* são os valores 2 e 3, o *operador* é o sinal de mais ('+'). O *resultado* dessa operação é 5. Damos o nome de *expressão* quando temos uma combinação de operações matemáticas (ou lógicas) envolvendo variáveis numa mesma linha de código.

Os *operadores matemáticos* em Python são os seguintes:

+ # adição (ou concatenação)
- # subtração
***** # multiplicação
/ # divisão
// # parte inteira de uma divisão: 10//3=3
% # resto de uma divisão: 10%3=1

Existe também os *operadores de comparação*. São eles:

== # igual (o símbolo '**=**' é um operador de atribuição de valores a variáveis)
!= # diferente
> # maior que
< # menor que
>= # maior ou igual
<= # menor ou igual

Por fim, temos os *operadores lógicos*:

not # negação ou inversor
and # verdadeiro se todas as condições forem também verdadeiras
or # verdadeiro se qualquer condição for verdadeira

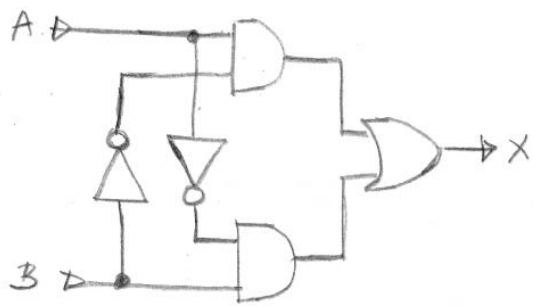
Os operadores lógicos ou de comparação dão resultados *booleanos*: *True* ou *False*. Assim, *2*==*3* é falso, pois 2 não é igual a 3. Da mesma forma, a operação *True* ***or*** *False* é verdadeira.

E o operador lógico '*xor*', ou *ou-exclusivo*? Sabemos que a porta lógica *xor* retorna '1' somente se ao menos uma das entradas for '1', não as duas. Para termos esse operador em Python podemos implementá-lo numa *função* usando operadores lógicos *and*, *or* e *not*, como na tela a seguir; ou importar uma biblioteca externa (*operator*) para dentro do Python. Mais adiante vamos aprender o que são bibliotecas em Python e como utilizá-las em nossos *scripts*.

```
1 def xor(x, y):
2     return bool((x and not y) or (not x and y))
3
4 print(xor(0,0))
5 print(xor(0,1))
6 print(xor(1,0))
7 print(xor(1,1))
```

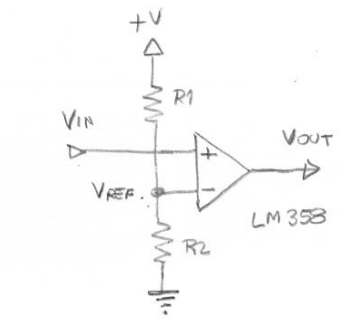
```
False
True
True
False
```

O circuito eletrônico equivalente dessa função *xor* com portas lógicas é o seguinte:



4- O Comparador de Tensão *While*

Comparadores normalmente são utilizados para diferenciar as duas condições *booleanas: Verdadeiro* ou *Falso*. Um circuito comparador de tensão montado com um LM-358, como o da figura a seguir, confronta as tensões em suas duas entradas e mostra em sua saída ou uma tensão alta (*High*) ou uma baixa (*Low*), conforme uma entrada seja maior que a outra. Um comparador desse tipo é essencialmente uma célula conversora A/D, já que, dependendo do nível de tensão analógica em uma entrada em relação à outra, tomada como referência, a saída do comparador vai ser '0' ou '1'.

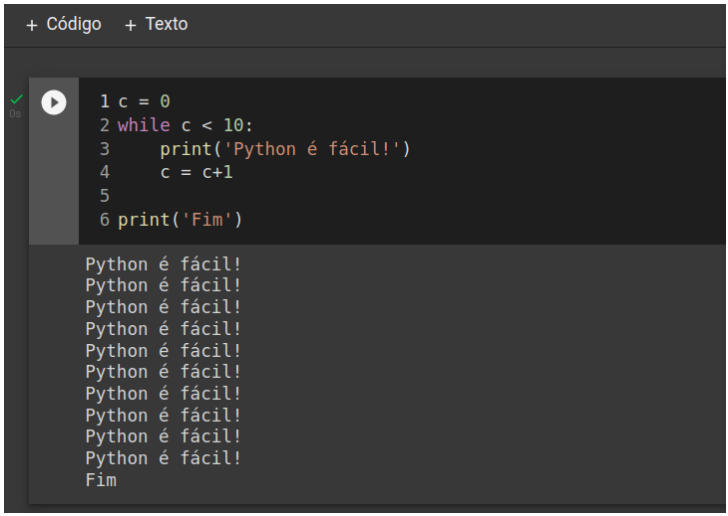


No Python temos uma estrutura de decisão, o laço *while*, que tem um comportamento parecido com um circuito comparador de tensão. Sua saída vai ser *True* ou *False* dependendo do resultado de uma operação matemática ou lógica na entrada. Existem no Python três estruturas de repetição, são elas: o laço '*FOR'*, o laço *'WHILE'* e a de decisão *'IF'*. Vejamos primeiro a sintaxe do laço *while*, que é a seguinte:

```
while expressao_teste:
    bloco_codigos
```

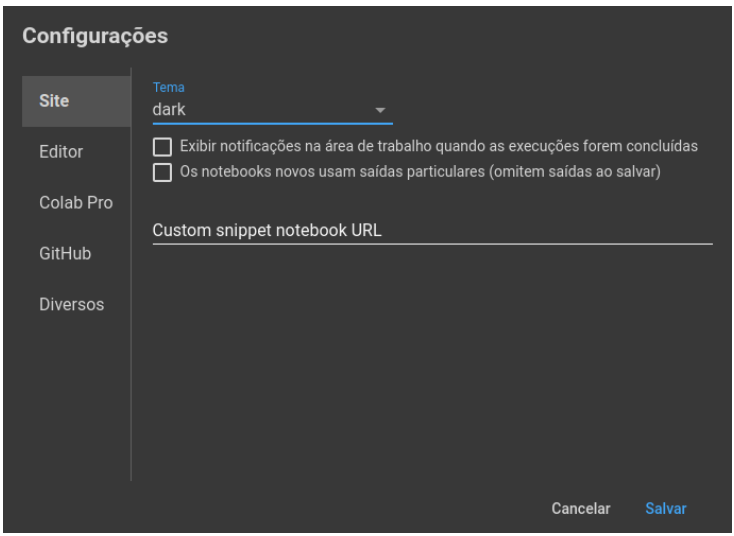
O laço *While* acima avalia a *expressao_teste* na primeira linha e verifica se o resultado da expressão é verdadeiro ou falso. Verdadeiro é qualquer resultado que não seja zero, portanto '1'; e falso um resultado zero ('0'). Somente se o resultado da expressão for verdadeiro as linhas do *bloco_codigos* serão executadas uma a uma; se falso a execução de todo o bloco é saltada, e o fluxo do programa segue normalmente com a execução das linhas depois do bloco.

Vamos entender isso melhor fazendo alguns experimentos com esse laço de decisão. Abra o *Google Colab* com seu *login* e *senha,* ou seu interpretador Python preferido, para testarmos o pequeno *script* listado a seguir.

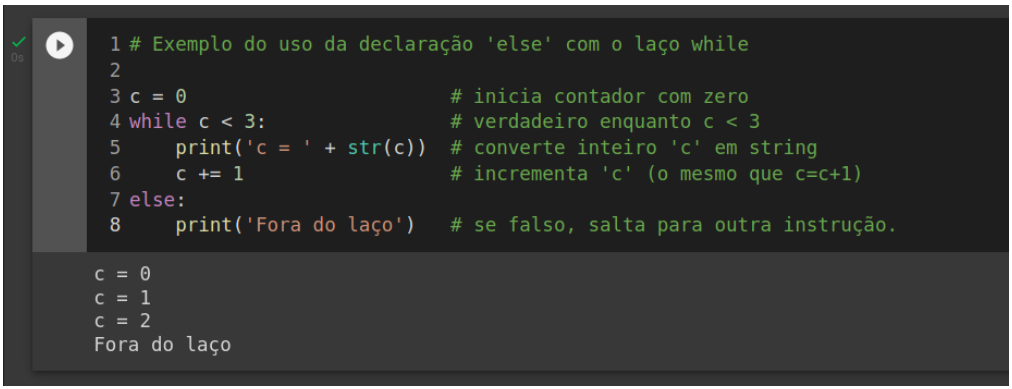


Na linha 1 criamos uma variável *'c'* (contador) e a iniciamos com '0'. Na linha 2 temos o laço *while* e a expressão *'c < 10'*, finalizada com dois pontos (':'). Repare que as linhas que formam o bloco de instruções são recuadas da margem esquerda 4 espaços (1 'Tab'). Nessas linhas, 3 e 4, colocamos as instruções que queremos repetir enquanto *(while)* a expressão *'c < 10'* for verdadeira, ou seja o texto *'Python é fácil!'* vai ser mostrado na tela do editor e o contador incrementado. Quando c==10, a expressão *'c < 10'* vai ser falsa e o laço quebrado. A próxima linha do *script,* fora do laço, mostra na tela a palavra 'Fim'.

Mudamos o tema do fundo da nossa plataforma *Colab* para *'dark'* na janela de *Configurações* no menu *Ferramentas*.



O laço *while* também tem uma declaração opcional *else*, como no laço de repetição *for*, que veremos depois. No *script* mostrado abaixo, enquanto c < 3, o resultado da expressão é verdadeiro e assim as instruções do bloco dentro do laço (linhas 5 e 6) vão sendo executadas. Se o resultado for falso, quando *c==3,* o resultado vai ser falso e o comando da execução passa para a linha indentada com *else*. Depois, todas as linhas abaixo, se houverem, serão executadas uma de cada vez.

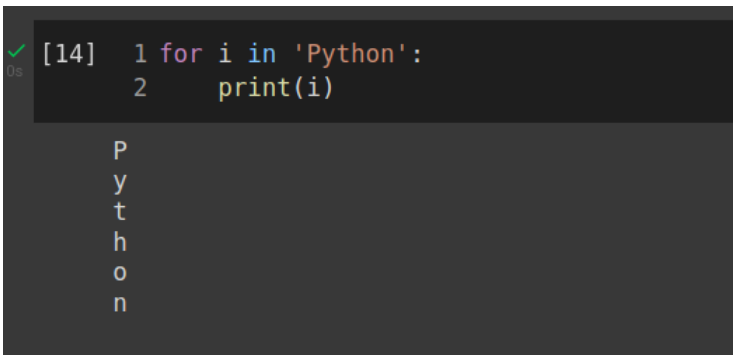


```
1 # Exemplo do uso da declaração 'else' com o laço while
2
3 c = 0                        # inicia contador com zero
4 while c < 3:                 # verdadeiro enquanto c < 3
5     print('c = ' + str(c))   # converte inteiro 'c' em string
6     c += 1                   # incrementa 'c' (o mesmo que c=c+1)
7 else:
8     print('Fora do laço')    # se falso, salta para outra instrução.
```

```
c = 0
c = 1
c = 2
Fora do laço
```

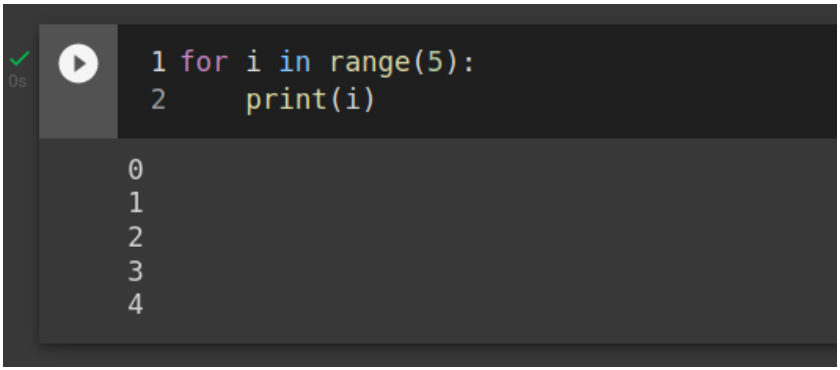
5- O laço de repetição *for* no Python

O laço de repetição *for* no Python executa as expressões do bloco indentado para cada item encontrado em um *objeto iterável*. Ok, vamos entender isso. Primeiro, tudo na linguagem Python é objeto. Variáveis são objetos; funções são objetos; operadores são objetos. E *iterável*, que vem de *iterable*, é um objeto composto por outros objetos menores e em que se pode iterar, ou tomar um a um cada objeto que compõe o objeto maior. Vamos para a prática: nos 3 exemplos mostrados abaixo, são *objetos iteráveis* a *string* (texto) 'Python', a função *range(5)* e a lista de componentes eletrônicos *['R1','R2','Q1','C1']*. A variável 'i' vai assumir o valor de cada item que compõe o objeto iterável. Tranquilo?



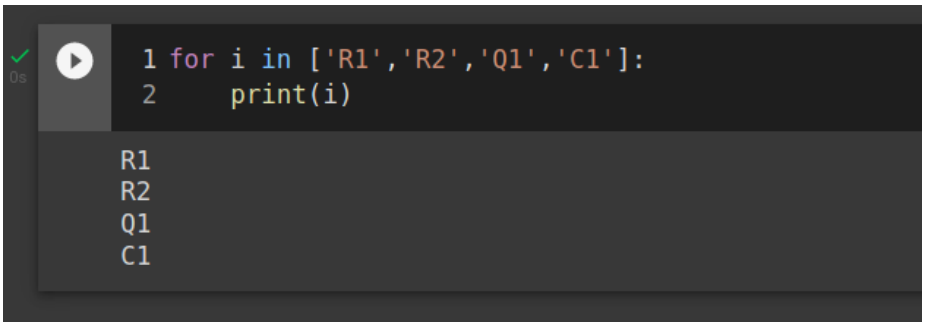
```
[14]  1 for i in 'Python':
      2     print(i)
```

```
P
y
t
h
o
n
```



```
1 for i in range(5):
2     print(i)
```

```
0
1
2
3
4
```



```
1 for i in ['R1','R2','Q1','C1']:
2     print(i)
```

```
R1
R2
Q1
C1
```

No primeiro exemplo a *string* (caracteres delimitados por aspas simples ou duplas) *'Python'* é decomposta em suas letras, que são mostradas uma em cada linha. No segundo temos a função *range()* que é uma *built-in function* muito usada no Python. Ela retorna uma sequência de números inteiros de 0 até o último número passado como parâmetro, mas subtraído de 1; no exemplo, de 0 a 4. No terceiro exemplo, o objeto iterável é uma *lista* de itens. Listas são estruturas de dados da linguagem Python que veremos somente no próximo capítulo dessa série.

6- Montando um Conversor A/D Virtual

Vamos agora juntar todos os comandos até agora vistos para montar um circuito*, em Python,* de uma célula conversora analógica-digital. Veja a listagem do programa abaixo.

Logo que o *script* é executado, ele pede ao usuário que entre com o valor de uma tensão de referência, que será atribuído à variável *Vref*, para o comparador de tensão, aqui o laço *while*. Por *default*, a função *input()* retorna uma *string*, assim, na mesma linha, convertemo-la para um número fracionário. Também atribuímos o valor '0' à variável *Vin*, a tensão analógica que queremos converter para digital.

```
Vin = 0                          # inicia variável Vin=0
Vref = float(input('Tensão de Referência: '))   # entrada do usuário
for i in range(10):
    while Vin < Vref:
        print('|', end =" ")     # troca mudança de linha por espaço
        Vin += 0.5               # incrementa Vin
    else:
        print('_', end =" ")
print()
print('Fim')
```

```
Tensão de Referência: 6
| | | | | | | | | | _ _ _ _ _ _ _ _ _ _
Fim
```

Dentro do laço *while*, enquanto a tensão de entrada for menor que a tensão de referência, um traço vertical é mostrado na tela do editor e a tensão de entrada é aumentada em 0.5 volts. Quando *Vin* ultrapassar *Vref*, na tela teremos uma série de traços horizontais. O laço *for* somente expande os traços horizontais em 10 posições.

Concluindo

Vamos fechar essa parte II de nossa série sobre a linguagem Python para Técnicos em Eletrônica com um pequeno programa em Python, um *Prediction Game*, para testar se você possui dons de *clarividência*. A máquina vai rolar um dado comum virtual e o usuário terá que *'adivinhar'* qual a face ficou para cima. São 10 dados rolados, um de cada vez, em cada sessão de teste; mas esse número pode ser mudado. Segundo *experts* em assuntos parapsicológicos, para ser validado esse teste teria que ser em 25 ou 50 rolagens.

```
import random                    # carrega biblioteca random
score = 0                        # contador de pontos ganhos

print('Vou jogar um dado e você terá que me dizer')
print('qual é a face do dado que ficou pra cima.')
print('São 10 tentativas. Pronto?')
for i in range(10):
    pc=(random.randint(1,6))     # o PC gera um Nº aleatório entre 1 e 6
    print(str(i+1) + 'ª tentativa:')
    hum = int(input('Qual é a face do dado?'))  # humano escolhe um Nº
    if pc == hum:
        score+=1                 # toda vez que o humano acertar soma 1
    elif hum > 6:
        print('Atenção: dados só tem até 6 faces!')     # erro!

if score >= 8:
    print('Caramba!... Você acertou '+str(score)+' vezes')
elif score >= 7:
    print('Parabéns: Você acertou '+str(score)+' vezes')
elif score >= 5:
    print ('Nada mal. Você acertou '+str(score)+' vezes')
else:
    print ('Você só acertou '+str(score)+' vezes.')
```

No código completo, mostrado acima, na primeira linha importamos uma biblioteca do Python, a biblioteca *random*.

Sobre bibliotecas: o núcleo do Python é o interpretador e também um conjunto de funções *batteries included*, prontas para uso; são as *built-in functions*, que formam a biblioteca básica da linguagem*;* algumas já conhecemos, como *input()*, *print()* e outras.

Mas existe outros conjuntos de funções, aqui chamados de *módulos*, criados por colaboradores do Python, sempre em expansão, que dão novas funcionalidades à linguagem.

A biblioteca *random* é um desses conjuntos de módulos que podemos importar, ou incorporar, aos nossos *scripts* em Python quando queremos gerar números aleatórios.

Na linha 2, o PC gera um número inteiro aleatório entre 1 e 6; as faces de um dado comum.

Depois de rolar o dado, ou seja, gerar um número qualquer entre 1 e 6, a máquina pede ao usuário que adivinhe qual o número que corresponde à face do dado que caiu para cima. Depois de 10 rolagens, o PC vai mostrar qual foi o resultado do teste.

Convido o leitor a tentar modificar esse programa para o modo *premonição*, quando o usuário tenta antever o número antes de ser gerado pela máquina.

Mãos à massa e até breve!

João Alexandre Silveira*

As portas lógicas *AND, OR* e *NOT* são consideradas como os blocos básicos de construção da Eletrônica Digital. É com combinações dessas portas que são montados todos os outros circuitos digitais mais complexos, dos *flip-flops* às *CPUs*.

Assim também podemos dizer dos *amplificadores operacionais* em relação à Eletrônica Analógica; essenciais para interfaces homem-máquina e sensores na internet das coisas.

Também podemos considerar que as células biológicas são os blocos básicos de construção dos seres vivos.

Então, também podemos dizer que toda linguagem de programação de computador tem seus blocos de construção básicos: são os ***controles de estrutura****,* que, basicamente, servem para alterar o fluxo contínuo de um programa; podemos arriscar e dizer que são quase como potenciômetros em circuitos de *software*.

Esses controles de estrutura, em certas situações, permitem ao programa tomar a decisão de seguir um caminho ou outro, pular uma parte de si mesmo ou repetir a execução de um mesmo bloco de comandos diversas vezes; eles quebram o fluxo normal linha a linha de um programa para decidir o que fazer a partir desse ponto.

*__Autor do livro “Experimentos com o Arduino”, disponível em__ www.amazon.com.br

Já vimos dois desses controles de estrutura na parte II dessa série sobre experimentos com Python: os laços de repetição *for* e *while*. O laço *for* executa os comandos no bloco indentado tantas vezes quantas for o número dado por um contador de eventos. Com este laço *for* podemos, por exemplo, executar um mesmo bloco de comandos de acordo com o valor armazenado numa variável enviada por um sensor; e depois seguir o fluxo normal do programa.

O laço *while* não tem contador, ele toma sua decisão de executar um bloco de comandos baseado no resultado de uma expressão e as variáveis nela contida: enquanto as variáveis mantiverem o resultado da expressão verdadeiro, o laço *while* fica executando repetidamente o bloco indentado; quando falso, interrompe a execução e deixa o fluxo do programa seguir seu caminho. Por exemplo: Vá incrementando a variável *var_1* enquanto *var_2 > var_3*.

Mais tarde, talvez possamos fazer experimentos com o quanto evoluímos desse comportamento para as máquinas aprenderem sozinhas, com dados e matemática. Seguindo em frente nessa parte III, vamos ver com mais detalhes o controle de estrutura *if...else* em Python, que vimos *en passant* na parte II.

Ainda na parte II, conhecemos a plataforma *on-line Google Colaboratory* (*Colab*) para testarmos nossos *scripts* em Python cada vez maiores. Vamos continuar com essa plataforma nos experimentos dessa parte III. Por fim, falamos dos operadores matemáticos, dos de comparação e dos lógicos. Os primeiros dão como resultado números inteiros ou fracionários. Os outros dois dão saídas *booleanas*: verdadeiro ou falso (‘1’ ou ‘0’). Existe também no Python o *operador de números complexos*, do tipo: *z = 3 + 2j*.

O CONTROLE DE ESTRUTURA IF...ELSE

O controle de estrutura *if* é parecido com o controle *while* visto antes; só que, aqui, se uma dada condição for atendida (for verdadeira) o bloco de comandos indentado vai ser executado *uma única vez*; depois o fluxo do programa segue normalmente. No *while* o bloco fica sendo executado *enquanto* essa condição for verdadeira: o fluxo do programa fica retido pelo controle de estrutura. Parece sutil a diferença, não? Se o resultado for falso, em ambos os controles, o bloco ou deixa de ser executado ou é interrompido. A sintaxe desse controle de estrutura *if* tem o seguinte formato:

```
if <expressão>:
    <comando_1>
    <comando_2>
    <comando_3>
```

Onde <*expressão*> é a condição que precisa ser verdadeira para que os comandos 1 a 3, indentados no controle *if*, sejam executados.

Vamos deixar de conversa e partir para os experimentos?

No *script* do *Prediction Game* que mostramos na parte II dessa série de artigos, usamos o controle *if* para comparar o número gerado randomicamente, correspondente à face virada para cima de um dado virtual, com aquele 'adivinhado' pelo experimentador e, depois, para comparar sua *performance* com uma tabela de valores. Vamos aproveitar a linha do gerador randômico daquele *script*.

Abra seu Colab (ou seu interpretador Python) e vamos digitar os comandos mostrados na tela abaixo.

O controle de estrutura *if* no *script* acima nos diz que se o número inteiro de 1 a 10 gerado pelo módulo *randint(),* da biblioteca *random* do Python, é um número par, checando se o resto da divisão desse número randômico por 2 é zero. Se o resto da divisão for zero, o número é par, senão (*else*) o número é ímpar. Toda vez que executamos esse *script,* na tela primeiro veremos qual foi o número inteiro gerado (linha 4); de acordo com o resultado da expressão *n_rand%2*, na linha 6, uma das funções *print()*, nas linhas 7 e 9, será executada uma vez. Após a execução do *script*, no exemplo dado, são mostrados o número 7 e a palavra '*ímpar'*.

Agora veja o quão elegante é a linguagem Python: podemos reescrever o *script* acima colocando o controle de estrutura *if...else* numa única linha. Veja na tela abaixo.

```
[81]  1 n_rand = random.randint(1,10)
      2 print(n_rand)
      3
      4 print('par' if n_rand % 2 == 0 else 'impar' )
```

```
4
par
```

Se queremos realizar mais de um teste de validação para a mesma expressão dentro do controle de estrutura *if*, usamos também o comando *elif*, uma aglutinação de *else* e *if*. Vejamos o exemplo na tela abaixo.

```
1 import random
2
3 n_rand = random.randint(1,10)
4 print('N° gerado: ' + str(n_rand))
5
6 if n_rand >= 5:
7     print('maior ou igual a 5')
8 elif n_rand <= 5:
9     print('menor ou igual a 5')
```

```
N° gerado: 4
menor ou igual a 5
```

Juntando tudo, temos a tela a seguir, onde vemos que dos 10 números inteiros gerados randomicamente pela função *randint()*, somente os dois primeiros números são rejeitados pelos filtros passa-banda – nas linhas 6 e 8 - codificados com controles de estrutura *if.*

```
1 import random
2
3 n_rand = random.randint(1,10)
4 print('N° gerado: ' + str(n_rand))
5
6 if n_rand > 5 and n_rand <= 9:
7     print('maior que 5 e menor ou igual a 9')
8 elif n_rand == 10 or n_rand > 2:
9     print('igual a 10 ou maior que 2')
10 else:
11     print('1 ou 2')
```

```
N° gerado: 9
maior que 5 e menor ou igual a 9
```

No *script*, o filtro da linha 6 só deixa passar os números inteiros maiores que 5 e menores ou iguais a 9. O filtro da linha 8 deixa passar o número 10 e aqueles maiores que 2. Ficam de fora dos filtros os números 1 e 2.

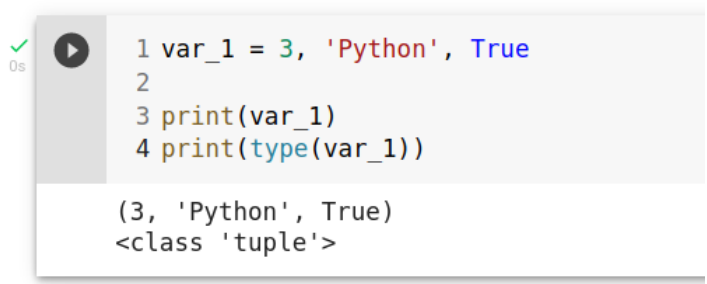
Vimos de forma ainda muito resumida que o controle de estrutura *if...elif...else*, também chamado de estrutura, ou declaração condicional, é utilizado para executar um bloco de comandos somente se, estabelecida uma certa condição, esta for atendida, se ela se mostrar verdadeira; e executar um outro bloco de comandos se essa condição não for atendida, se ela for falsa. No decorrer dessa nossa série sobre experimentos com Python vamos ver o quão versátil é esse controle quando estivermos desenhando *scripts* mais complexos.

TUPLAS EM PYTHON

A *Física Newtoniana* sempre nos ensinou que dois corpos não podem ocupar um mesmo espaço físico. Essa asserção nos parece óbvia se concluímos que a matéria bruta é constituída por montagens inteligentes de átomos, que se combinam espacialmente mas não se entrelaçam nunca. Essa individualidade atômica vem de uma força maior que mantem essa estrutura atômica: a interação das forças e torques das principais partículas que formam o núcleo (prótons e nêutrons) e, ao seu redor, dos elétrons em suas órbitas elípticas.

Nessa linha, não poderíamos guardar diferentes valores numa única variável, num mesmo espaço em uma memória física semicondutora dentro de um computador. Mas parece que existem coisas que não seguem à risca esses preceitos da *Física Clássica;* existe algo como ***superposição de estados***.

Bom, os parágrafos acima foram somente uma provocação quântica ao leitor, para falarmos de uma estrutura de dados na linguagem Python conhecida por *Tupla* (ou *tuple* em inglês), onde em uma só variável podemos guardar vários valores. Só que aqui o Python cria espaços físicos na memória do computador para conter todos os valores declarados. Vejamos a tela abaixo.

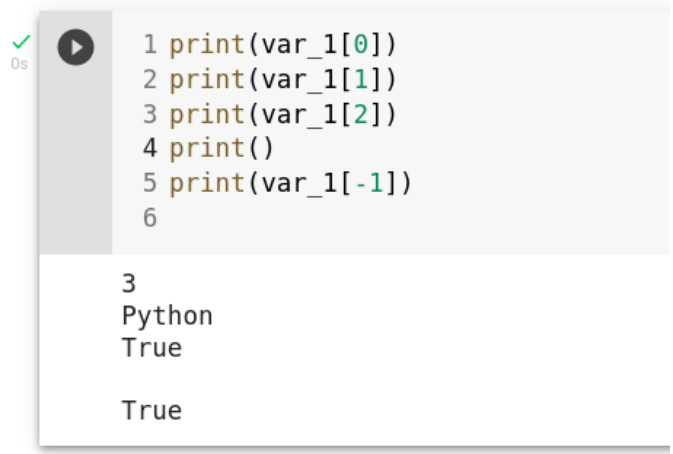


Observe que guardamos os valores 3, 'Python' e True numa mesma variável var_1: um número inteiro, uma string e um número booleano. Quando checamos o conteúdo dessa variável com a função *print()*, notamos que o Python nos mostra os três valores atribuídos à variável *var_1* dentro de parênteses. Isso é uma tupla. Uma estrutura de dados da linguagem que também podemos chamar de *variável composta*. Na parte I desta nossa série conhecemos as variáveis simples: *containers* onde guardamos um só valor, de qualquer tipo: numérico, fracionário, *string, booleano* e outros.

Variáveis compostas são listas de qualquer tamanho que guardam valores de qualquer tipo. São como containers com divisões internas separadas por vírgulas. No caso da *tuplas*, esses valores vem sempre entre parênteses. Nas listas propriamente ditas (*Lists*) do Python, que veremos mais abaixo, esses valores vem sempre entre colchetes (‘*[* ‘ e ‘*]*).

E como selecionamos cada um desses valores na tupla *var_1*?

Simples: chamando a tupla pelo seu nome junto com o seu índice relativo, a posição ‘física’ do item que queremos selecionar, colocado entre colchetes. O índice do primeiro elemento é 0 (zero). Assim, se queremos conhecer o primeiro valor na tupla acima, digitamos *var_1[0]*. O segundo *var_1[1], e assim por diante*, como mostrado na tela abaixo. O último elemento de qualquer tupla é dado pelo índice [-1], na tupla do exemplo é *var_1[-1]*.



Podemos conhecer o índice de qualquer elemento em uma tupla com o método *index()*. Por exemplo, se queremos saber qual é o índice do elemento ‘Python’ na tupla *var_1* acima, digitamos: *print(var_1.index(‘Python’))*.

Uma vez criada uma tupla, seus elementos não podem ser mudados, ou seja: não podemos adicionar outros itens ou remover qualquer um deles. Por segurança nas informações, usamos tuplas quando não queremos que nenhum dado nelas armazenados sofra qualquer mudança por outras partes do nosso *script*. Agora, veja só que interessante: vamos declarar duas variáveis *var_1* e *var_2* de duas formas, atribuindo-lhes o número inteiro 10:

```
var_1 = 10
var_2 =10,
```

A diferença entre as duas declarações é somente a vírgula depois do valor atribuído à variável *var_2*. A primeira é uma variável simples; a segunda é uma tupla com um só elemento. Experimente digitar no seu editor de *scripts* o comando *print(type(var_1))* e *print(type(var_2))* para ver qual o tipo de cada uma dessas variáveis.

Mais adiante vamos entender tudo isso melhor.

LISTAS EM PYTHON

Listas (*Lists* em inglês) em Python são outro tipo de variáveis compostas. São estruturas de dados criadas como uma variável qualquer; e nelas guardamos, não um, mas uma coleção de diferentes objetos, como números, *strings*, outras listas, também tuplas e todo tipo de expressões matemáticas ou lógicas.

Pensemos numa lista de mercado: é uma estrutura lógica vertical onde anotamos todos os itens que precisamos comprar, um embaixo do outro. Nessa lista os itens são separados por linhas. As listas em Python são estruturas horizontais, onde os elementos nelas contidos ficam numa mesma linha, porém separados por vírgulas, como nas tuplas, mas delimitadas por colchetes (‘[‘ e ‘]’). Uma pequena lista de mercado poderia ser escrita como uma lista em Python, dessa forma:

```
lst_mercado = [ ‘feijão’, ‘arroz’, ‘batata’, ‘tomate’, ‘frango’ ]
```

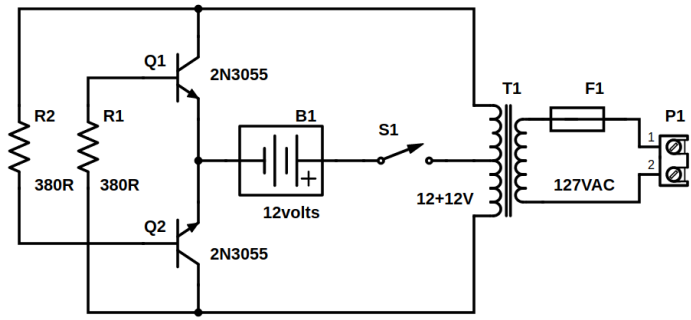
Todos os 5 itens dessa lista *lst_mercado* estão dentro de colchetes e separados por vírgulas. Se não usássemos essa estrutura de variável composta, teríamos que criar 5 variáveis simples, uma para cada item da nossa lista. Podemos criar listas vazias, estruturas declaradas antecipadamente para posterior uso, que podem ser declaradas no Python de duas formas:

```
lst_0 = [ ]      #usando colchetes sem nenhum dado
lst_0 = list()       #usando o método list() da linguagem
```

A principal diferença entre tuplas e listas é que as últimas podem ser modificadas por comandos dentro de nossos *scripts* e as tuplas não. E, para isso, existem métodos próprios como *append(), insert(), remove()* e outros, que não são aceitos pelas tuplas.

Vejamos o diagrama eletrônico abaixo, um clássico inversor CC/CA com dois transistores 2N3055 formando um oscilador harmônico do tipo *push-pull*: cada transistor conduz alternadamente em sua seção do enrolamento primário do transformador, induzindo uma onda simétrica no secundário. Esse oscilador é bastante empregado em inversores de baixa frequência por entregar bem mais potência que os outros tipos de circuitos osciladores. Vamos, então, criar uma lista com as *strings* que identificam cada componente no circuito:

lst_id = ['S1', 'R1', 'R2', 'Q1', 'Q2', 'T1']



Como nas tuplas, podemos acessar qualquer item dessa lista somente declarando seu nome e a posição relativa, ou índice, do item desejado entre colchetes; como na tela *Colab* abaixo:

```
1 lst_id = [ 'S1', 'R1', 'R2', 'Q1', 'Q2', 'T1' ]
2
3 print(lst_id)
4 print(type(lst_id))
5
6 print(lst_id[0])        #primeiro item
7 print(lst_id[2])        #terceiro item
8 print(lst_id[5])        #sexto item
9 print(lst_id[-1])       #ultimo item
```

```
['S1', 'R1', 'R2', 'Q1', 'Q2', 'T1']
<class 'list'>
S1
R2
T1
T1
```

Nessa tela acima, depois de criada a lista, primeiro checamos quais são seus elementos, depois confirmamos com a função *type ()* que essa estrutura é mesmo de uma lista. A partir da linha 6 acessamos os itens 0, 2 e 5, e o último item da lista.

Também podemos tomar 'fatias' de uma lista, como na tela a seguir:

```
1 print(lst_id[:3])       #seleciona do primeiro ao terceiro item
2 print(lst_id[3:])       #seleciona do quarto ao último item
3 print(lst_id[1:4])      #seleciona do segundo ao quarto item
4 print(lst_id[:])        #seleciona todos os itens da lista
5
```

```
['S1', 'R1', 'R2']
['Q1', 'Q2', 'T1']
['R1', 'R2', 'Q1']
['S1', 'R1', 'R2', 'Q1', 'Q2', 'T1']
```

Vejamos, agora, alguns dos métodos que já vem embutidos (*built-in*) nessa estrutura de dados *list* do Python: *insert(i, x), append(x), remove(x), count(x), copy(), clear(), sort() e reverse()*. Veja na tela abaixo:

```
[81]  1 lst_id = [ 'S1', 'R1', 'R2', 'Q1', 'Q2', 'T1' ]
      2
      3 lst_id.insert(0,'B1')   #inclui 'B1' na posição 0 da lista
      4 lst_id.append('P1')     #inclui 'P1' na última posição da lista
      5
      6 print(lst_id)
```

```
['B1', 'S1', 'R1', 'R2', 'Q1', 'Q2', 'T1', 'P1']
```

```
[82]  1 lst_id.remove('B1')         #remove a primeira ocorrencia de 'B1'
      2 print(lst_id.count('R1'))   #conta quantos 'R1' existe na lista
      3
      4 print(lst_id)
```

```
1
['S1', 'R1', 'R2', 'Q1', 'Q2', 'T1', 'P1']
```

```
[83]  1 lst_id_copia = lst_id.copy()    #faz uma copia da lista
      2 print(lst_id_copia)
      3
      4 lst_id_copia.clear()            #remove todos os itens da lista.
      5 print(lst_id_copia)
```

```
['S1', 'R1', 'R2', 'Q1', 'Q2', 'T1', 'P1']
[]
```

```
1 lst_id.sort()       #itens em ordem crescente
2 print(lst_id)
3
4 lst_id.reverse()    #itens em ordem decrescente
5 print(lst_id)
```

```
['P1', 'Q1', 'Q2', 'R1', 'R2', 'S1', 'T1']
['T1', 'S1', 'R2', 'R1', 'Q2', 'Q1', 'P1']
```

Vamos criar mais duas listas, uma com os nomes dos componentes eletrônicos para montar o circuito do inversor CC/CA e uma lista vazia:

lst_comp = ['trafo', 'chave', 68, 68, '2N3055', '2N3055', 'terminal']
lst_inversor = [ ]

E, no *Colab,* vamos inserir com o método *append()*, as listas *lst_id* e *lst_comp* na lista vazia que criamos. Observe que em *lst_comp* temos itens repetidos, sendo dois deles que não são *strings*, mas números inteiros.

Já é sabido que podemos criar listas com quaisquer objetos, inclusive com tuplas ou outras listas.

```
lst_comp = [ 'trafo', 'chave', 68, 68, '2N3055', '2N3055',  'terminal' ]
lst_inversor = [ ]

print(lst_id)
print(lst_comp)
print(lst_inversor)
print()
lst_inversor.append(lst_id)
lst_inversor.append(lst_comp)
print(lst_inversor)

```

```
['T1', 'S1', 'R2', 'R1', 'Q2', 'Q1', 'P1']
['trafo', 'chave', 68, 68, '2N3055', '2N3055', 'terminal']
[]

[['T1', 'S1', 'R2', 'R1', 'Q2', 'Q1', 'P1'], ['trafo', 'chave', 68, 68, '2N3055', '2N3055', 'terminal']]
```

Agora temos duas listas dentro de uma outra lista. O primeiro item dessa nova lista é a lista *lst_id*, o segundo *lst_comp*. Como acessamos qualquer elemento de uma das listas dentro da lista *lst_inversor*? Simples: chamamos a lista principal seguido do índice da lista que queremos e depois o índice do item nessa lista interna.

Confuso? Vamos ao Colab:

```
[102] print(lst_inversor[0])          #lista de índice 0 na lista principal
      print(lst_inversor[1])          #lista de índice 1 na lista principal
```

```
['T1', 'S1', 'R2', 'R1', 'Q2', 'Q1', 'P1']
['trafo', 'chave', 68, 68, '2N3055', '2N3055', 'terminal']
```

```
print(lst_inversor[0][0])       #primeiro item da primeira lista
print(lst_inversor[1][2])       #terceiro item da segunda lista
print(lst_inversor[1][-2])      #penúltimo item da segunda lista
```

```
T1
68
2N3055
```

Listas em Python é um tema muito interessante mas extenso; são muitos os métodos e funções disponíveis na linguagem para se tratar com seus itens. Lá na frente, veremos que podemos montar matrizes com listas de dados, os chamados dataframes. Vejamos mais alguns exemplos:

```
lst = list(range(1,11))                #lista com primeiros 10 inteiros
s1 = sum([j for j in lst if j%2 == 0])  #soma dos N°s pares
s2 = sum([k for k in lst if k%2 != 0])  #soma dos N°s impares

print(lst)
print(s1)
print(s2)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
30
25
```

Nesse script, criamos na primeira linha uma lista com 10 números inteiros; nas duas linhas seguintes criamos duas variáveis, s1 e s2: uma para guardar a soma dos números pares e outra para guardar a soma dos números ímpares.

A essa estrutura dá-se o nome de ***list comprehension***, algo como '*compreensão de listas'* em português, no sentido de faculdade de conter em si; uma forma concisa de criar listas a partir de uma outra lista. Nos próximos capítulos dessa série de artigos, vamos ter a oportunidade de explorar bastante essa facilidade do Python.

CONCLUINDO

Leonardo Fibonacci foi um matemático italiano da cidade de Pisa que no século XII descobriu uma série matemática que leva seu nome, a *série de Fibonacci,* também conhecida por *série áurea*, que parece estar presente em muitas formas geométricas da natureza, do molusco marinho nautilus às galáxias no cosmos.

Nessa série cada número é igual à soma dos dois números precedentes. Vamos criar uma lista com Python com os 12 primeiros números da série de Fibonacci:

```
fibo = [0,1]                #lista inicial Fibonacci
[fibo.append(fibo[-2]+fibo[-1]) for i in range(10)]

print(fibo)
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Nessa tela iniciamos uma lista com dois elementos (0 e 1) e a ela vamos adicionando com o método *append()* a soma dos dois números anteriores por 10 vezes.

A seguir, temos uma tabela resumida com algumas funções e métodos que já testamos com listas no Python:

insert() insere um novo item numa dada posição *#fibo.insert(0,'Fibonacci')*
remove() remove um item da lista *#fibo.remove('Fibonacci')*
pop() remove um item numa dada posição *#fibo.pop(-1)*
len() retorna a quantidade de elementos numa lista *#len(fibo)*
'+' concatena duas listas numa única *#fibo + list(range(90,95))*
min() retorna o menor valor numérico na lista *#min(fibo)*
max() retorna o maior valor numérico na lista *#max(fibo)*
sum() retorna a soma de todos os valores numéricos na lista *#sum()*
sort() organiza os elementos da lista em ordem crescente *#fibo.sort()*
reverse() organiza os elementos em ordem inversa *#fibo.reverse()*
count() retorna o número de ocorrências de um elemento *#fibo.count(1)*

Fechamos esta parte III com um *script* bem simples que simula o conhecido jogo com moedas cara ou coroa.

```
1 import random                          #importa biblioteca random
2 i, ca, co = 0,0,0                      #inicia variáveis com 0
3
4 while i < 100:
5     m_bool = random.choice(['cara', 'coroa'])   #moeda lançada
6     if m_bool == 'cara': ca += 1       #contador de caras
7     elif m_bool == 'coroa': co += 1 #contador de coroas
8     i += 1                             #contador de moedas
9
10 #resultado:
11 print('Caras: ' + str(ca))
12 print('Coroas: ' + str(co))
13 print('Diferença: ' + str(abs(ca-co)))
```

```
Caras: 48
Coroas: 52
Diferença: 4
```

Nesse script, o método choice() da biblioteca random retorna randomicamente um elemento da lista ['cara','coroa'] na linha 5; e a função abs() na linha 13 retorna o valor absoluto da diferença entre o número de caras e coroas, ou seja, ela remove o eventual sinal negativo do resultado. Até breve!

João Alexandre Silveira*

Chegamos à quarta etapa dessa nossa trilha a um objetivo: aprender Python fazendo experiências com Eletrônica. Até aqui somente com *software*, mas logo vamos integrar também alguns circuitos eletrônicos à linguagem.

Vamos em frente, pois tão importante quanto o caminho, é o caminhante e o caminhar.

Até aqui vimos os comandos principais de qualquer linguagem de programação moderna: os laços de repetição *for*, *while* e *if*; e duas estruturas de dados: as *tuplas* e as *listas*.

Nessa parte IV vamos conhecer mais uma estrutura de dados: os *dicionários* em Python. Os dicionários são também variáveis compostas, onde podemos guardar conjuntos de diferentes objetos em Python.

E como tudo na linguagem Python é objeto, veremos que num dicionário podemos guardar qualquer tipo de números, *strings*, variáveis simples, tuplas, listas e até outros dicionários.

Depois, vamos conhecer as bibliotecas em Python; são métodos e funções prontas, disponíveis por desenvolvedores Python na forma de código aberto, para quase qualquer área de aplicação.

*__Autor do livro "Experimentos com o Arduino", disponível em__ www.amazon.com.br

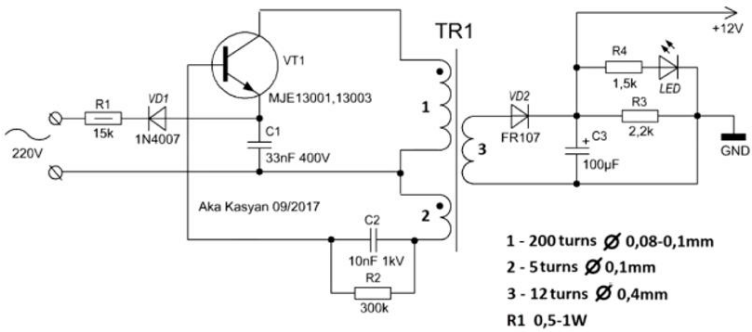
Dicionários em Python

Dicionários na linguagem Python são, como as tuplas e as listas, também estruturas de dados na forma de *containers* com divisões internas. Nas tuplas e nas listas essas divisões internas têm como índice fixo para cada elemento armazenado um número inteiro. Nos dicionários, esse índice, em vez de um número, pode ser uma *string*, uma sequência de caracteres alfanuméricos. Assim fica muito mais fácil ir buscar um valor em uma estrutura desse tipo por um nome, como nas variáveis simples, em vez de pôr um número. Com isso conseguimos personalizar os índices dos itens nessa estrutura de dados.

Ainda estamos falando de formatos de listas, estruturas de dados lineares, como uma lista de compras de mercado. Mais adiante vamos combinar essas listas em objetos bidimensionais: os **dataframes** no Python, tabelas como uma planilhas tipo Excel. *En passant:* existem métodos na linguagem Python para ler e escrever diretamente em planilhas Microsoft Excel.

Já vimos na parte III dessa série, que as tuplas são identificadas por seus elementos virem entre parênteses - '(' e ')' - e as listas entre colchetes - '[' e ']'. Os dicionários em Python são identificados por seus elementos virem entre chaves - '{' e '}'.

Agora vejamos o circuito da figura abaixo. Trata-se de um inversor CA/CC muito simples que achamos na *internet*: uma fonte de alimentação, conhecida como *fonte chaveada* (*smps-switched mode power supply*), sem o clássico transformador de entrada. São dois estágios isolados: à esquerda, um oscilador *Hartley*, formado pelo transistor Q1 e o duplo enrolamento primário de T1, é alimentado com alta tensão retificada e filtrada diretamente da rede AC de 127/220 volts. Essa alta tensão oscilatória em alta frequência induz no secundário de T1 uma tensão menor mas com alta corrente. O estágio da direita é também uma clássica fonte retificadora de meia-onda formada pelo secundário de T1 e D2-C3. R3 é o resistor de carga e o LED com seu limitador de corrente R4 serve para indicar que temos tensão de saída. Veja no [youtube] essa fonte funcionado.



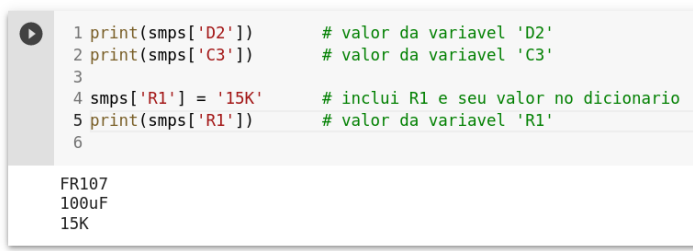
Vamos então criar um dicionário em Python para guardar somente 6 dos componentes do circuito da nossa fonte chaveada. Veja na tela do Colab abaixo. Demos o nome de *smps* ao nosso dicionário, e, entre suas chaves, colocamos as informações correlacionadas em pares, separados por vírgulas.

Veja que *smps* é um dicionário Python onde guardamos a identificação de cada componente no diagrama elétrico e seu correspondente valor ou tipo. Aqui, cada item do dicionário é formado por um par *key:value* (chave:valor). Os elementos de cada par são separados por dois pontos (‘:’). No nosso dicionário smps o primeiro par é ‘Q1’:’MJE13001’, o último é ‘C3’:’100uF’.

Podemos listar os nomes de todas as variáveis (as chaves) de nosso dicionário smps com o método *keys()* do dicionário Python: *smps.keys()*. Para conhecer somente os valores guardados nessas variáveis usamos o método *values()*: *smps.values()*. Se queremos saber tudo o que existe no dicionário, usamos o método *items()*: s*mps.items()*.

A chave é o nome da variável simples dentro da variável composta; valor é… o valor guardado nessa variável simples. No dicionário *smps* criado acima, as chaves são os índices, os nomes das variáveis simples, que vem antes dos dois pontos (‘:’); os valores das variáveis vêm depois dos dois pontos.

Depois de criado o dicionário, digamos que queremos saber somente o valor de alguns componentes e depois incluir mais um que não estava listado. Veja a tela Colab a seguir.

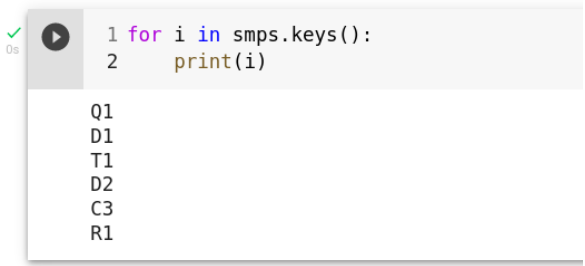


Também podemos remover qualquer item do nosso dicionário com o comando *del* da biblioteca básica do Python:

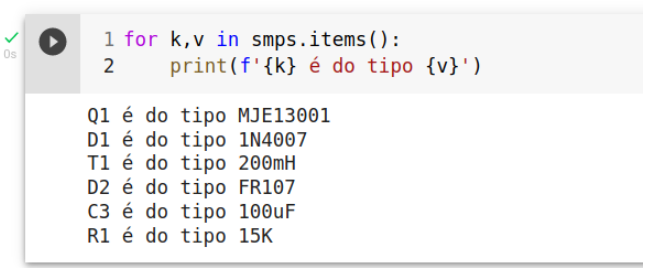
```
[12]  1 del smps['C1']      # exclui C1 do dicionário
      2 print(smps)         # ver dicionario atualizado
```

```
{'Q1': 'MJE13001', 'D1': '1N4007', 'T1': '200mH', 'D2': 'FR107', 'C3': '100uF', 'R1': '15K'}
```

Podemos listar todas as chaves (nomes das variáveis) do nosso dicionário com o laço de repetição *for*, com na tela abaixo. Experimente também: *smps.values().* Se formos testar *smps.items()* com esse laço *for* vamos obter uma lista de tuplas.



Mas, se o método *items()* do dicionário Python nos fornece uma lista de tuplas com os nomes das variáveis e seus valores, podemos então fazer uma iteração direta nessa lista e criar uma nova lista mais inteligível para humanos, utilizando iteradores no laço de repetição *for* da seguinte forma:



O que temos nesse laço de repetição *for* no *script* acima? Esse laço atribui a cada par de variáveis *k-v* a chave e o valor do par *key-value* (uma tupla) em *smps.items()*; depois mostra na tela uma listagem com as identificações e os valores dos componentes do circuito da nossa fonte chaveada.

Essa forma do laço *for* nos dicionários é semelhante àquela com *for...enumerate()*, já vista por aqui.

Vamos agora mudar na nossa listagem de componentes o valor do capacitor C3 de 100uF para 470uF. Veja no *script* a seguir que basta indicar junto ao nome do dicionário, a chave (a variável) que queremos mudar e o novo valor dela.

Também mostramos como formatar um texto com dados para a tela com a agora função *print()* do Python. Para formatar um texto dessa forma, inicie o conteúdo da linha que será mostrada na tela, entre os parênteses, com a letra minúscula *f* (de *format*). Coloque tanto o texto quanto os dados trazidos de variáveis dentro de um par de aspas simples; só que esses dados devem ser ainda colocados dentro de chaves '{' e '}'. Importante: essas chaves dentro da função *print()* fazem parte da sintaxe dessa função, não são dicionários!

```
smps['C3'] = '470uF'

for k,v in smps.items():
    print(f'{k} é do tipo {v}')
```

```
Q1 é do tipo MJE13001
D1 é do tipo 1N4007
T1 é do tipo 200mH
D2 é do tipo FR107
C3 é do tipo 470uF
R1 é do tipo 15K
```

Podemos criar listas com os dados nos dicionários passando como parâmetros esses dados para a função *list()* do Python, como podemos na tela seguinte.

```
lst_k = list(smps.keys())       # lista de variaveis
lst_v = list(smps.values())     # lista de valores
lst_i = list(smps.items())      # lista de tuplas com variaveis e valores

print(lst_k[0])                 # primeiro elemento da lista de variaveis
print(lst_v[4])                 # quinto elemento da lista de valores
print(lst_i[-1][0])             # primeiro elemento da ultima tupla.
```

```
Q1
FR107
C3
```

Agora, só para brincar com dicionários em Python, vamos criar um novo dicionário ainda com os componentes da nossa fonte chaveada. E criar um texto com as informações que temos nos dois dicionários.

Veja a tela Colab abaixo.

```
[12]  1 smps_2 = { 'Q1':'NPN Alta Freq.',
      2          'D1':'Retif 700V',
      3          'C1':'400V',
      4          'T1':'Toroidal',
      5          'D2':'Retif rapido',
      6          'C3':'25V'
      7           }
      8
      9 print(smps_2)

{'Q1': 'NPN Alta Freq.', 'D1': 'Retif 700V', 'C1': '400V', 'T1': 'Toroidal', 'D2': 'Retif rapido', 'C3': '25V'}

  1 print(f'O transistor {list(smps.keys())[0]} tipo {list(smps.values())[0]} é um {list(smps_2.values())[0]}')

O transistor Q1 tipo MJE13001 é um NPN Alta Freq.
```

Vamos desenhar um *script* um pouco mais interativo? Copie e execute o código na tela a seguir no editor da sua plataforma Colab.

```
[21]  1 dic_1 = dict()      # ou dic_1 = {}
      2 lst_1 = list()
      3
      4 for i in range(4):
      5     dic_1['ID'] = input('Componente?: ')    # identificacao do componente
      6     dic_1['Valor'] = input('Valor?: ')      # valor
      7     dic_1['Preço'] = input('Preço?: ')      # preço
      8
      9     lst_1.append(dic_1.copy())              # adiciona novo dicionario na lista
     10     #print(lst_1)

Componente?: q1
Valor?: mjf13001
Preço?: 1.5
Componente?: t1
Valor?: toroide
Preço?: 2.2
Componente?: d2
Valor?: fr107
Preço?: 1.3
Componente?: r3
Valor?: 2k2
Preço?: 0.15
```

Começamos criando um dicionário *dic_1* e uma lista *lst_1* vazios (linhas 1 e 2 no código Python no Colab). Na linha 4 temos um laço *for* que vai pedir ao usuário que entre pelo teclado do PC com 3 informações para somente 4 componentes do circuito: uma identificação para o componente, um valor e um preço – linhas 5, 6 e 7. Essas 3 informações vão ser guardadas nas chaves 'ID', 'Valor' e 'Preço' do dicionário *dic_1*. Portanto o dicionário *dic_1* pode guardar até 3 pares casados de dados para cada componente. Na linha 9 uma cópia desse dicionário *dic_1* vai ser adicionado a lista *lst_1*.

Na última linha do *script* você pode remover o sinal de comentário (*hash '#'*) antes da função *print()* e observar na tela como a cada iteração do laço a lista *lst_1* vai aumentando de tamanho.

Só para relembrar como acessar qualquer elemento numa lista Python, vimos isso na parte III dessa série, experimente 'pinçar' uma informação de um componente num dos dicionários da lista *lst_1* testando os comandos da tela Colab a seguir.

```
[31]  1 print(lst_1[0])                 # tudo do primeiro componente
      2 print(lst_1[2]['Valor'])        # valor do terceiro componente
      3 print(lst_1[-1]['Preço'])       # preço do ultimo
```

```
{'ID': 'q1', 'Valor': 'mjf13001', 'Preço': '1.5'}
fr107
0.15
```

Se o leitor observar a lista *lst_1* depois de todas as entradas dos dados dos 4 componentes da fonte chaveada, verá que os quatro dicionários (um para cada componente) lá guardados têm em comum os nomes das chaves: 'ID', 'Valor' e 'Preço'. Somente seus valores são diferentes.

Pensemos: se colocarmos as 3 chaves comuns desses 4 dicionários em 3 colunas, seus valores ficariam em 4 linhas. O que temos aí? Claro: uma matriz bidimensional! Veja na tela abaixo como ficaria essa matriz com os dados nos dicionários da lista *lst_1*.

	ID	Valor	Preço
0	q1	mje13001	1.5
1	t1	toroide	2.2
2	d2	fr107	1.4
3	r3	2k2	0.15

OK, mas *matrizes* e *dataframes* (*datasets*) são temas para um novo artigo inteiro, e vai depender de fazermos importações de algumas bibliotecas para dentro de nosso compilador Python. E é esse o papo a seguir: bibliotecas no Python.

Mas, antes, veja o resumo das principais operações que podemos fazer com os métodos dos dicionários Python, na tabela abaixo.

clear()	remove todos os itens do dicionário	*# dic_1.clear()*
get()	retorna o valor de uma chave no dicionário	*# dic_1.get('Preço')*
items()	retorna uma lista de tuplas com os pares k-v	*# dic_1.items*()
keys()	retorna uma lista das chaves no dicionário	*# dic_1.keys*()
values()	retorna uma lista com os valores no dicionário	*# dic_1.values*()
pop()	remove uma chave do dicionário	***# dic_1.pop('ID')***

Bibliotecas em Python

Antes dos computadores pessoais existiam as calculadoras eletrônicas pessoais. Todo mundo tinha a sua; até relógios digitais de pulso vinham com teclados de calculadoras, umas muito simples, para cálculos básicos, e outras bem especializadas. Para cálculos financeiros tinha a *HP-12C e p*ara cálculos mais avançados tinha a *HP-41C*, ambas da *Hewlett-Packard*.

Já existiam calculadoras que aceitavam módulos, cartões de memória *ROM*, comprados separadamente para áreas específicas, como química e geologia, que eram inseridos num compartimento próprio na calculadora; da mesma forma como inserimos um módulo USB ao nosso PC hoje. A *TI-59* da *Texas Instruments* era uma dessas calculadoras. Era como trocar cartucho no videogame *Atari 2600*.

Esses cartuchos, ou melhor, módulos, eram bibliotecas que agregavam novas funções à calculadora. Essas bibliotecas eram conjuntos de pequenos programas já prontos voltados para áreas, como dissemos, específicas, e que só podiam ser executados naquele modelo de calculadora.

A linguagem Python aplica esse conceito de bibliotecas. O núcleo do Python já vem com todas as funções básicas como qualquer outra moderna linguagem de programação, mas, no Python, é possível acrescentar novas funções à linguagem através da importação de bibliotecas para dentro do núcleo.

Lá na segunda parte dessa nossa série de artigos já fizemos uso de uma biblioteca Python. Foi a biblioteca *random*, da qual usamos a função *randint()* para gerar os números inteiros das faces de um dado virtual.

A tabela com os componentes eletrônicos da página anterior foi criada com um módulo de uma biblioteca importada chamada *pandas*. Tudo o que fizemos, depois de importar a biblioteca, foi chamar o módulo *DataFrame()* dessa biblioteca e passar para ele a lista *lst_1* com os dicionários que contêm as informações sobre os componentes da fonte chaveada. Fica mais claro vendo a tela Colab abaixo.

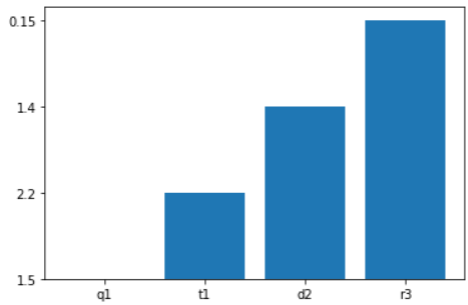
```
[21]  1 import pandas as pd
      2
      3 df_1 = pd.DataFrame(lst_1)
      4 print(df_1)
```

```
   ID     Valor Preço
0  q1  mje13001   1.5
1  t1   toroide   2.2
2  d2     fr107   1.4
3  r3       2k2  0.15
```

Na primeira linha importamos toda a biblioteca *pandas* e a chamamos de *pd*, para simplificar o código. Depois, numa única linha chamamos o módulo *DataFrame()* dentro da biblioteca *pandas* e lhe passamos como parâmetro toda a lista *lst_1*. O resultado guardamos na variável *df_1*, para depois exibi-la na tela.

Vejamos mais uma demonstração de uma das bibliotecas Python, talvez a mais requisitada pelos desenvolvedores Python para gerar gráficos: a biblioteca *matplotlib*. Veja a tela Colab a seguir.

```
[33]  1 import matplotlib.pyplot as plt
      2
      3 plt.bar(df_1['ID'], df_1['Preço'])
      4 plt.show()
      5
```



Na primeira linha importamos como *plt* o módulo *pyplot()* de plotar gráficos da biblioteca *matplotlib*. Depois mandamos plotar na forma de barras, diretamente do dataframe *df_1*, os preços dos componentes. Não se assuste, isso é só uma degustação, todas essas bibliotecas ainda vamos ter a oportunidade de estudá-las mais a fundo nos trabalhos futuros.

Uma biblioteca bastante interessante é a *yfinance*, uma ferramenta de código aberto que permite baixarmos os dados de negociações de preços e volume das ações nas bolsas de valores ao redor do mundo. Esses dados são apanhados diretamente do site da *yahoo!finance*. Se uma biblioteca não faz parte do conjunto padrão da linguagem Python, antes de importá-la precisamos instalá-la no nosso PC. É o caso da biblioteca *yfinance* e grande maioria das bibliotecas de finanças e científicas. Conforme formos avançando em nossos experimentos com Python, iremos mostrar como instalar e importar partes ou toda uma biblioteca para o nosso PC. Veja na tela a seguir como se comportaram os papeis da Petrobras nos primeiros meses deste ano.

```
[13]  1 petr4_df = yfinance.download('PETR4.SA',
      2                      start='2022-01-01',
      3                      end='2022-03-31',
      4                      progress=False,
      5                      )
      6 print(petr4_df.tail())
```

```
                Open       High        Low      Close  Adj Close     Volume
Date
2022-02-24  34.799999  35.290001  32.680000  33.389999  33.389999  139674100
2022-02-25  33.450001  34.000000  32.900002  34.000000  34.000000   86189100
2022-03-02  35.259998  35.290001  34.389999  34.669998  34.669998   58071800
2022-03-03  34.820000  34.930000  34.160000  34.240002  34.240002   69237400
2022-03-04   0.000000   0.000000   0.000000  34.230000  34.230000          0
```

Outras duas bibliotecas interessantes são a *Requests* e a *BeautfulSoap*. A primeira nos permite fazer requisições de páginas *web* (documentos *HTML*) diretamente da rede digital mundial, como se fosse um usuário humano comum diante de seu PC. Nesses documentos *HTML* podemos fazer uma "raspagem" (*scraping*) e coletar somente dados importantes de tabelas e textos.

E tudo isso não é nada complicado: veja o *script* na tela abaixo, rodando num outro editor de linguagens: a plataforma *Spider*. Aqui capturamos a principal manchete do portal G1 da Globo, que era a seguinte no dia e hora que rodamos o *script*:

Com o método *get()* da biblioteca *requests* requisitamos na web a página principal do portal G1. Com o método *find()* da biblioteca *BeautfulSoup* encontramos a manchete no documento *HTML* recebido. De posse dessa informação, podemos criar um outro *script-robô* para enviar para nosso *e-mail* a lista de notícias desse portal, digamos, a cada 3 horas.

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 import requests
5 from bs4 import BeautifulSoup
6
7 response = requests.get('https://g1.globo.com/')
8 content = response.content
9 soup = BeautifulSoup(content, 'html.parser')
10
11 noticia = soup.find('div', attrs={'class':'feed-post-body-title gui-color-primary gui-color-hover'})
12 print(noticia.text)
13
```

```
In [12]: runfile('/home/johnny/Desktop/ANTENNA/MAR_2022/g1_scraping_07mar22.py', wdir='/home/johnny/Desktop/
ANTENNA/MAR_2022')
CENTRAL GLOBONEWS traz tudo da 3ª rodada de negociações entre Rússia e Ucrânia

In [13]:
```

Também muito interessante é a biblioteca *Scikit-learn*, que é uma coleção de ferramentas para criação de algoritmos para *aprendizado de máquinas* (*machine learning*), um ramo da hoje tão propalada *Inteligência Artificial*. Tem grande aplicação no descobrimento de padrões ocultos em grandes volumes de dados, como é o caso de séries históricas de preços de papéis negociados em bolsas de valores.

A lista de bibliotecas para a linguagem Python é imensa, todas disponíveis gratuitamente para desenvolvedores como nós. Aqui vai uma pequena coletânea de algumas das bibliotecas que temos usado bastante.

Pandas	biblioteca para tratamento de grandes volumes de dados
Numpy	para pesquisa em computação científica e cálculos com matrizes
Matplotlib	criação de todo tipo de gráfico 2D ou 3D
Requests	usada para requisições de documentos HTML na web
BeautfulSoap	a mais utilizada em *web scraping* (coleta de dados na web)
Scipy	para matemáticos e afins, com módulos de álgebra linear e estatística
Scikit-Learn	criação de algoritmos para aprendizado de máquina
Tkinter	conjunto de ferramentas para criação de interfaces gráficas

Links de fontes citadas no artigo:

"Experimentos com o Arduino" - ebook do autor, à venda na Amazon.com.br

https://www.youtube.com/watch?v=qaHXIrQFPYk - montagem da fonte chaveada

https://www.electro-tech-online.com/threads/oscillator-in-a-simple-smps.158509/ - diagrama

Até breve!

João Alexandre Silveira*

Sejam bem-vindos de novo à nossa série sobre experimentos com a linguagem Python para técnicos, engenheiros, inventores e hobistas de Eletrônica. Cremos que, até aqui, vimos a essência dessa interessante linguagem. Na parte IV discorremos sobre dicionários e bibliotecas em Python. Lá, vimos que os dicionários são identificados por seus elementos virem entre chaves - '{' e '}'; e por seus índices poderem vir na forma de *strings*, uma sequência de caracteres alfanuméricos, e não por somente números inteiros, como nas tuplas e listas. Também falamos que bibliotecas são coleções de módulos e funções prontas que importamos para dentro do núcleo do Python, para acrescentar novas funções à linguagem.

Nesta quinta parte, vamos ver o que são erros de sintaxe, que podem aparecer quando estamos escrevendo um código em Python, e também como tratamos as exceções, que podem ocorrer quando executamos um programa nessa linguagem.

Lá no primeiro artigo desta nossa série, em dezembro do ano passado, dissemos que *código fonte* é a versão primeira de um programa de computador. É uma abstração materializada, uma lista com instruções para uma máquina programável que devem obedecer às regras de sintaxe da linguagem de programação escolhida. Essa lista, quase sempre escrita em inglês, contém todas as tarefas que queremos que tal máquina execute, como, por exemplo, tomar dois números como entradas guardados em duas variáveis, somar seus conteúdos e exibir o resultado numa tela. Temos aqui o que se costuma chamar de *Algoritmo*; como uma receita culinária onde devemos obedecer a todas as instruções listadas para se chegar a um resultado desejado.

*__Autor do livro "Experimentos com o Arduino", disponível em__ www.amazon.com.br

O código fonte, após passar pelo interpretador (ou compilador) da linguagem escolhida, gera o *código objeto*, o programa final, a linguagem montada com bits e bytes que o computador entende.

Vamos fazer uma analogia com o projeto de um circuito eletrônico: procurando uma solução eletrônica para uma necessidade que temos, primeiro mentalizamos de modo muito objetivo as funções que nosso circuito deverá ter. Então rascunhamos numa folha de papel os blocos básicos desse circuito que idealizamos. Cada bloco poderá vir a ser um circuito funcional que deverá se comunicar com os circuitos dos outros blocos; e assim vamos abstraindo até chegarmos a um circuito único e o mais simples possível. Normalmente o projeto da fonte de alimentação é a última etapa do nosso projeto.

Por fim, montamos todo ou em partes o circuito final; e testamos fisicamente cada estágio numa *protoboard*, ou de modo virtual num simulador de circuitos em nosso PC. É nessa fase, da prototipagem, que os problemas (as falhas ou erros no projeto) podem surgir; como componentes eletrônicos (resistores, capacitores, transistores e outros) mal dimensionados ou com defeito, incompatibilidade entre partes do próprio circuito, e tantos outros.

Assim também é quando estamos desenvolvendo um programa de computador: a partir de uma necessidade, a mãe da invenção, segundo o filósofo Platão, abstraímos uma possível solução; agora não com componentes eletrônicos, mas com componentes da sintaxe da linguagem de programação que dominamos. Aqui, igualmente, desenhamos num papel os blocos básicos de construção do programa na forma de um fluxograma e depois os abrimos em seus componentes funcionais menores: as variáveis, os operadores lógicos e matemáticos, os laços de repetição, as funções e as bibliotecas que precisamos importar. Tudo isso já vimos nos artigos anteriores.

Depois de montar todo o sistema num editor de textos (a nossa '*protoboard*'), ou numa plataforma de desenvolvimento integrada (*IDE*), como o Colab que até aqui temos usado, é hora de testar cada parte ou o todo no interpretador (ou no compilador) da linguagem escolhida. É aqui que também vão aparecer as falhas e os erros de projeto do nosso programa, que precisamos corrigir. Esses erros também podem ser de qualquer natureza, desde o mais comum, como erros de sintaxe e acesso a memória com dados corrompidos, a conexão remota incompatível ou ausente, ou mesmo entradas de informações incorretas pelo usuário.

Quase todas as linguagens modernas têm seus próprios métodos de correção de erros, chamados de *depuração* (*debuging*, em inglês). O Python tem um sistema robusto de detecção desses erros e falhas durante a compilação (translação) do *código fonte* em *código objeto*; ou depois da compilação, quando da execução do programa.

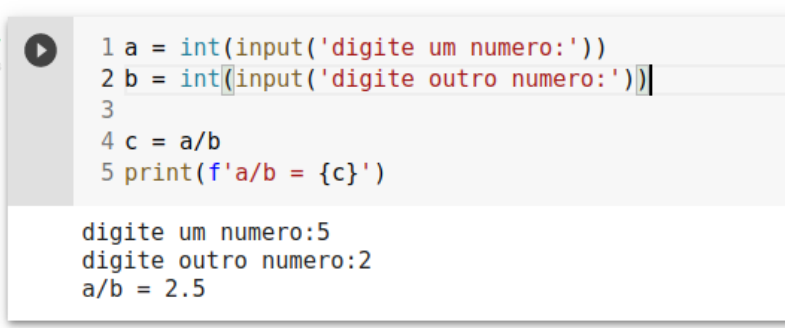
Os erros que são detectados somente durante a execução do programa recebem o nome de *exceções* (*exceptions*), quando então o programa trava e uma mensagem de texto indicando o erro é enviada ao programador. Por exemplo, o usuário entra com algum dado incorreto, como um denominador 0 em uma operação matemática de divisão.

Exceções são também erros, mas que só são detectados quando o programa está sendo executado, pois interrompem o fluxo normal do programa (*program crash*); mas muitos deles podem ser prevenidos se tratados antecipadamente ainda na fase do projeto, no código fonte. Esse tratamento de possíveis erros podem incluir alertas quando da entradas incorretas pelo usuário (como no caso da divisão por zero); estabelecimento de valores *default* se o usuário omitir uma entrada de dado obrigatória; ou um salto (*jump*) para um bloco de comandos alternativo.

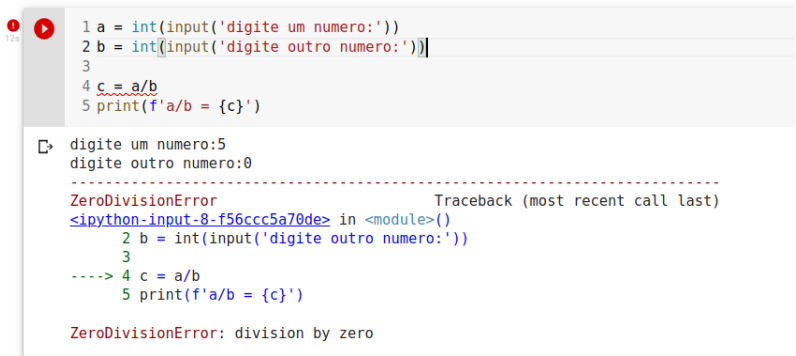
Os Erros e Falhas na criação de *Scripts* em Python

Uma falha muito comum que podemos cometer durante a criação do código fonte, principalmente quando iniciamos numa nova linguagem de programação, é o erro de sintaxe; quando o comando que digitamos não é reconhecido pelo compilador do Python. Pode acontecer também quando não obedecemos ao estilo da linguagem ou esquecemos de incluir alguma parte essencial da estrutura de um comando. Os erros de sintaxe não podem ser prevenidos e tratados com alertas ou com valores *default*, como nas exceções; aqui a compilação do código fonte é interrompida e também uma mensagem de erro (*SyntaxError*) é gerada para alertar o programador.

OK, mas vamos ver tudo isso no Colab. Primeiramente, vamos ver alguns exemplos de erros de programação bastante comuns em *scripts* Python. Comecemos com um *script* bem simples como o mostrado na tela abaixo.



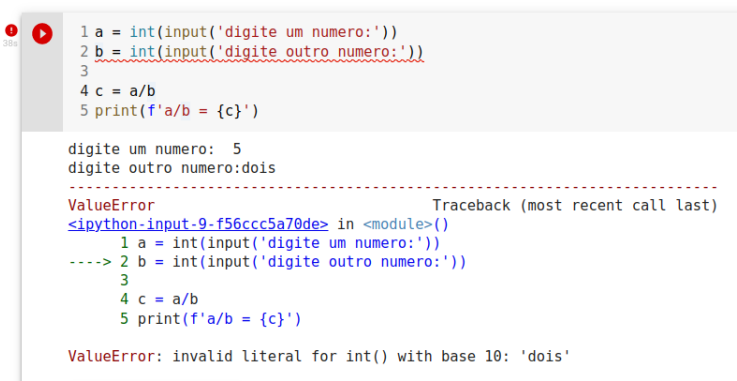
Nessa tela temos um *script* que deve mostrar o resultado da divisão entre dois números inteiros. A primeira linha requisita um número ao usuário e guarda o seu valor na variável 'a'. A segunda linha requisita outro número e o guarda na variável 'b'. Na linha 4 o programa divide o primeiro número pelo segundo e guarda o resultado na variável 'c'. Por fim, na linha 5, o *script* mostra o resultado dessa operação na tela do computador. Até aqui nenhuma surpresa, o resultado é 2.5 como vemos nessa primeira tela do Colab. Note que o resultado é um número fracionário. Agora veja a segunda tela, abaixo.



```
1 a = int(input('digite um numero:'))
2 b = int(input('digite outro numero:'))
3
4 c = a/b
5 print(f'a/b = {c}')
```

```
digite um numero:5
digite outro numero:0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-8-f56ccc5a70de> in <module>()
      2 b = int(input('digite outro numero:'))
      3
----> 4 c = a/b
      5 print(f'a/b = {c}')

ZeroDivisionError: division by zero
```

Nessa segunda tela, entramos para o segundo número, o denominador, com o valor 0. Sabemos que não é possível a divisão de qualquer número por zero; por isso quando o programa tenta fazer essa operação matemática na linha 4, o erro de entrada do usuário é detectado e uma exceção é mostrada logo a seguir, sem mesmo chegar à execução da linha 5, que deveria mostrar o resultado na tela. O compilador Python aponta, depois da linha tracejada vermelha, para onde ocorreu a exceção (seta tracejada verde): na linha 4, quando ele tentava dividir o primeiro número entrado pelo segundo. O tipo de erro também é indicado na última linha da tela: *ZeroDivisionError* e ainda explicita a seguir que se trata de um erro de divisão por zero. Legal!, parece fácil encontrar erros num código fonte Python, não? Mas, vamos continuar observando a tela seguinte, ainda com o mesmo *script* que começamos.



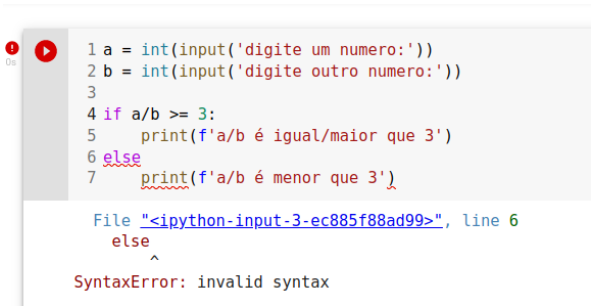
```
1 a = int(input('digite um numero:'))
2 b = int(input('digite outro numero:'))
3
4 c = a/b
5 print(f'a/b = {c}')
```

```
digite um numero:  5
digite outro numero:dois
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-f56ccc5a70de> in <module>()
      1 a = int(input('digite um numero:'))
----> 2 b = int(input('digite outro numero:'))
      3
      4 c = a/b
      5 print(f'a/b = {c}')

ValueError: invalid literal for int() with base 10: 'dois'
```

Nessa terceira tela estamos tentamos dividir um número inteiro por uma *string*. Aqui também o compilador Python detectou uma exceção quando o *script* requisitou na linha 2 um número inteiro e entramos com a *string 'dois'*. O programa foi travado nessa linha, como podemos observar com a indicação da seta verde; as linhas seguintes sequer foram executadas. Aqui temos uma exceção chamada de *ValueError*: não se pode fazer operações matemáticas entre tipos diferentes, aqui um inteiro com uma *string*. Lembre-se de que exceções são os erros que só são detectados quando o programa já está sendo executado.

O Python não pode 'adivinhar' por si só se o usuário vai digitar o tipo correto exigido pelo programa. Mas veremos mais adiante que o programador do código fonte, este sim, pode tentar prever uma entrada errada pelo usuário e, se antecipando ao fato, caso ocorra uma entrada errada, criar uma mensagem de alerta pedindo ao usuário a entrada correta, sem travar o programa em execução.

Continuando, vamos incluir um laço de repetição *if...else* no *script* que temos no Colab, agora forçando intencionalmente erros de sintaxe muitos comuns. Veja a tela a seguir.



Tente o leitor descobrir, observando o alerta do Python, onde está o erro. Dica: Veja o apontador de erros: o sinal circunflexo que aparece na mensagem.

Como no exemplo anterior, o programa nos pede para entrar com dois números inteiros; mas agora tenta nos dizer se o resultado da divisão de um pelo outro é maior ou igual a 3. Quando tentamos compilar esse *script* um erro de sintaxe ocorre na linha 6. Veja que o compilador nos aponta onde está o erro: esquecemos de colocar os dois pontos exigidos pelo laço *if...else* logo depois do *else*. Corrija o erro no código e o execute de novo o *script* para o ver funcionando normalmente.

Outros erros comuns durante a criação do código fonte são: pri*m*t() - com 'm' em vez de 'n'; esquecer o recuo de 4 espaços (*Tab*) na estrutura de laços de repetição; não converter para número inteiro com a função *int()* a entrada *string* na função *input()*; fazer uma operação lógica ou matemática sem declarar uma das variáveis; operações lógicas ou matemáticas entre tipos diferentes; tentar usar uma biblioteca sem antes importá-la.

A lista é imensa e se fôssemos tentar mostrar todos os exemplos de erros de sintaxe aqui, esse artigo teria dezenas de páginas; somente errando e acertando com a prática da linguagem poderemos dirimir todos os erros em nossos *scripts* em Python. Veja no link [2] no final do artigo alguns exemplos de erros comuns de sintaxe.

Fazendo o Tratamento das Exceções

Voltemos ao código da página anterior e vamos fazer uma prevenção a um possível erro de entrada pelo usuário para o segundo número, o denominador, na operação de divisão entre inteiros. Veja a tela Colab a seguir.

```
[9]  1 a = int(input('digite um numero:'))
     2 b = int(input('digite outro numero:'))
     3
     4 try:
     5     if a/b >= 3:
     6         print(f'a/b é igual/maior que 3')
     7     else:
     8         print(f'a/b é menor que 3')
     9 except:
    10     print('Zero no denominador não é permitido!')

digite um numero:0
digite outro numero:2
a/b é menor que 3
```

Aqui o programa foi compilado e executado sem erros; se entrarmos com as entradas requisitadas corretas nenhuma exceção é detectada. Entramos com 0 para o primeiro número, o numerador, e com um número inteiro qualquer para o segundo, o denominador, o resultado da divisão foi também 0. Matemática pura. Mas, já sabemos que para qualquer número no numerador e 0 no denominador teremos um erro; no Python, isso será uma exceção que vai travar a execução normal do programa, o que não desejamos que aconteça.

Então, para prevenir essa possível exceção adicionamos três linhas ao nosso código fonte original. Vamos testar esse novo *script* entrando com 0 para o denominador. Veja a tela a seguir.

```
[10]  1 a = int(input('digite um numero:'))
      2 b = int(input('digite outro numero:'))
      3
      4 try:
      5     if a/b >= 3:
      6         print(f'a/b é igual/maior que 3')
      7     else:
      8         print(f'a/b é menor que 3')
      9 except:
     10     print('Zero no denominador não é permitido!')

digite um numero:5
digite outro numero:0
Zero no denominador não é permitido!
```

O que fizemos de especial nesse *script*? Nada mudamos quanto às posições das variáveis ‘a’ e ‘b’; mas colocamos o laço de repetição *if...else* do *script* original dentro de um bloco chamado *try;* e por prevenção, criamos um alerta para o usuário dentro de um bloco chamado *except*. O código dentro do primeiro bloco está ali para ser executado normalmente; mas é vulnerável a exceções dependendo da entrada de dados pelo usuário. Caso ocorra mesmo uma exceção, será o código dentro do segundo bloco é que será executado, não o do primeiro. Tranquilo?

Escrever códigos para computadores é uma forma de arte. Observe o *script* a seguir e o compare com o anterior.

```
[11]  1 a = int(input('digite um numero:'))
      2 b = int(input('digite outro numero:'))
      3
      4 try:
      5     c = a/b
      6 except:
      7     print('Zero no denominador não é permitido!')
      8 else:
      9     if c >= 3: print(f'a/b é igual/maior que 3')
     10     else: print(f'a/b é menor que 3')

digite um numero:5
digite outro numero:0
Zero no denominador não é permitido!
```

Qual é a diferença? Essa é o que poderíamos chamar de forma mais *pitônica* de escrever esse *script*. Pitônica? Sim, *forma pitônica* significa código que não apenas *obedece à sintaxe correta*, mas que também segue as convenções da *comunidade Python* e usa a linguagem da maneira que deve ser usada. Nesse código, primeiro testamos dentro do bloco *try* se a operação aritmética de divisão entre dois números não gera nenhuma exceção. Se sim, o comando dentro do bloco *except* é executado; se não, esse bloco é ignorado e o bloco dentro do primeiro *else* que é executado. Confuso? Então, vamos a mais alguns exemplos para deixar as coisas mais claras.

Na parte III de nossa série estudamos as listas em Python. Vimos que listas são variáveis compostas onde podemos guardar não somente um único valor, mas coleções de diferentes objetos; como números inteiros ou fracionários, *strings* de qualquer tamanho, outras listas com outras listas dentro; também tuplas e todo tipo de expressões matemáticas ou lógicas. Vamos criar uma lista simples com somente 4 objetos, componentes de um circuito eletrônico qualquer. Veja a tela a seguir.

```
[1]  1 lst_1 = [R3, C4, L1, Q2]
     2 print(lst_1)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-1e805d8b673b> in <module>()
----> 1 lst_1 = [R3, C4, L1, Q2]
      2 print(lst_1)

NameError: name 'R3' is not defined
```

Ôpa!, cometemos um erro. Criamos uma lista com variáveis simples sem declarar anteriormente nenhuma delas. Vamos corrigir nosso código.

```
[2]  1 R3, C4, L1, Q2 = 10K, 47uF, 10mH, 2N2222A
     2
     3 lst_1 = [R3, C4, L1, Q2]
     4 print(lst_1)
```

```
  File "<ipython-input-2-935b61f7446c>", line 1
    R3, C4, L1, Q2 = 10K, 47uF, 10mH, 2N2222A
                     ^
SyntaxError: invalid syntax
```

Putz!, erramos de novo. Agora atribuímos valores às variáveis mesclando números inteiros com *strings*. Vamos tentar mais uma vez.

```
[3]  1 R3, C4, L1, Q2 = '10K', '47uF', '10mH', '2N2222A'
     2 ''
     3 lst_1 = [R3, C4, L1, Q2]
     4 print(lst_1)
```

```
['10K', '47uF', '10mH', '2N2222A']
```

Agora sim! Criamos uma lista em Python com quatro objetos *string*; e mandamos imprimir essa lista na tela. Agora vamos checar somente o quarto elemento de nossa lista.

```
1 R3, C4, L1, Q2 = '10K', '47uF', '10mH', '2N2222A'
2
3 lst_1 = [R3, C4, L1, Q2]
4 print(lst_1[4])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-6-631237fcb09b> in <module>()
      2
      3 lst_1 = [R3, C4, L1, Q2]
----> 4 print(lst_1[4])

IndexError: list index out of range
```

Essa era a exceção que queríamos mostrar: um erro de índice (*IndexError)*. Estamos tentando acessar em nossa lista um elemento que não existe: um quinto elemento. Sabemos que o primeiro elemento de qualquer lista ou tupla em Python começa com o índice 0; assim o correto acesso ao quarto elemento dessa lista deveria ser: *lst_1[3]*.

Podemos criar, somente como exercício, uma prevenção contra esse tipo de exceção. Vejamos o *script* na tela abaixo.

```
[21]  1 R3, C4, L1, Q2 = '10K', '47uF', '10mH', '2N2222A'
      2 lst_1 = [R3, C4, L1, Q2]
      3
      4 i = int(input('escolha um item na lista: '))
      5
      6 try:
      7     lst_1[i]
      8 except IndexError:
      9     print(f'O item {i+1} não existe')
     10 else:
     11     print(f'O item {i} na lista é {lst_1[i]}')
     12 finally:
     13     print(f'A lista tem {len(lst_1)} itens')

escolha um item na lista: 4
O item 5 não existe
A lista tem 4 itens
```

Veja que introduzimos mais um bloco ao nosso *script*: o bloco *finally* que, indiferentemente da ocorrência ou não de exceções quando da execução do nosso programa, vai executar o código nele contido. A função *len()* nos informa quantos itens temos em nossa lista, e já foi por nós aqui estudada.

Concluindo

Lembre-se de que os erros de programação em Python são detectados na compilação do programa; exceções são detectadas na execução do programa e, também, que o bloco *try* testa um código por possíveis exceções; o bloco *except* trata a ocorrência dessas exceções.

Como vimos até aqui, aprender Python é fácil, talvez porque essa linguagem não seja 'engessada'; existem muitas maneiras de escrever um mesmo *script.* O programador é livre para escolher a forma que mais lhe convém; desde que, claro!, respeite sua sintaxe; e preferencialmente a escreva de forma pitônica.

Até breve!

Links:

[1] Livro "Experimentos com Arduino": procure por este título em https://www.amazon.com.br/
[2] Pequena lista de erros em Python: https://www.tutorialsteacher.com/python/error-types-in-python

João Alexandre Silveira*

Bem-vindos, caros leitores, novamente, a mais um capítulo de nossa série de experimentos com a linguagem Python para técnicos e engenheiros em Eletrônica. Como dissemos desde o início, nossa proposta com estes artigos mensais é tentar mostrar ao pessoal de *hardware* que fazer montagens com objetos de *software,* além de divertido, é tão fácil quanto montar um circuito eletrônico. Todos nós da Eletrônica já sabemos que os principais objetos usados para uma montagem eletrônica são resistores, capacitores, transistores, indutores e circuitos integrados.

E, agora, também sabemos que os principais objetos para a montagem de um *script* (programa) em Python são as variáveis, a estrutura condicional *if* e os laços de repetição, os operadores lógicos e aritméticos, as funções, as listas, as tuplas e também os dicionários e as bibliotecas. Tudo isso já vimos nos quatro primeiros artigos.

Nesta nossa série não iremos tratar de programação orientada a objetos (*OOP – Object-Oriented Programming*). O assunto daria uma nova série de artigos um tanto longa. Mas podemos adiantar o que é um *objeto*, o que é uma *classe* e uma *instância* em Python, conceitos básicos da programação *OOP*.

Vamos lá. Pense num instrumento de medição elétrica (analógico ou digital), como uma caixa retangular média de cor amarela, que contêm um mostrador na parte superior e uma chave seletora no centro de uma escala logo debaixo do mostrador. Mais embaixo vêm três bornes de cores distintas onde se pode conectar dois cabos das mesmas cores. Tudo o que descrevemos acima pode ser os *Atributos* ou *propriedades* de um instrumento de medição elétrica.

*__Autor do livro “Experimentos com o Arduino”, disponível em__ www.amazon.com.br

Continuando, este nosso aparelho tem a funcionalidade de, por exemplo, mensurar a grandeza de alguns fenômenos elétricos, como voltagens e correntes AC ou DC e determinar o valor de uma resistência à passagem da corrente elétrica. Também podemos testar a continuidade elétrica entre dois pontos e podemos até incluir medição de capacitâncias e de baixas frequências.

Agora estamos falando dos *Métodos* do instrumento de medição que concebemos, um *Objeto* conhecido por multímetro, o eterno companheiro do eletrônico!

Atributos e *Métodos*. Esses são dois conceitos da programação *OOP* relacionados aos objetos que o programador Python pensa em criar. Os atributos são sempre substantivos: tamanho, cor, peso, textura. Os métodos são sempre verbos: medir, aferir, multiplicar, testar. Todas as modernas linguagens de programação permitem dessa forma ao programador modificar estruturas já criadas, customizando seus atributos e métodos, e, assim, conseguir novas funcionalidades.

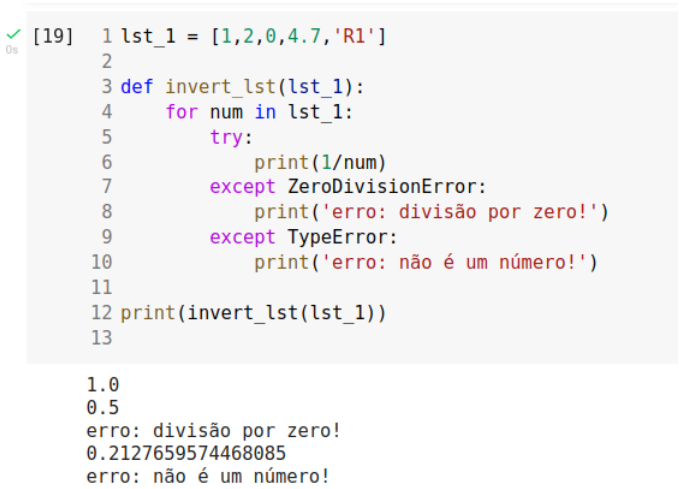
O instrumento de medidas elétricas tomado acima como exemplo, com mostrador, escala com chave seletora e ponteiras, pertence a uma classe de aparelhos usados por profissionais da área de Eletrônica: a *classe* multímetro. O instrumento amarelo que idealizamos é uma *instância* (objeto) da *classe multímetro*. São instâncias de uma classe objetos cujos comportamentos e estados são definidos pela classe. Conhecendo os *atributos* e *métodos* da *classe* multímetro, podemos criar, por exemplo, um novo instrumento que possa detectar ínfimas variações na corrente elétrica que podem surgir na interação de campos elétricos com substâncias radioativas. Entendidos esses conceitos, podemos ir adiante.

Mais experimentos com Erros & Exceções em Python

No capítulo 5 discorremos sobre como tratar os erros e as exceções que sempre podem ocorrer quando da montagem de um *script* usando a linguagem Python. Dissemos que os erros quase sempre são detectados durante a *compilação* (transformação) do código fonte escrito pelo programador, segundo a sintaxe da linguagem em código objeto, somente inteligível pelo processador da máquina. Já as exceções vimos que podem aparecer quando o *script,* depois de compilado, é executado.

Naquele artigo até fizemos experimentos com *scripts* para detectar exceções causadas quando da divisão de um número por zero; quando passamos para uma função uma *string* em vez de um número; e quando tentamos acessar numa lista um item que não existe. Vale a pena reforçar esse entendimento sobre erros e exceções com mais alguns exemplos, antes de tratarmos do tema do artigo desse mês: leitura e escrita de arquivos texto em Python.

Só para relembrar o que já vimos na parte 5, vamos de novo tratar essas exceções citadas acima numa única função em Python. Veja a tela *Colab* abaixo.



```
[19]  1 lst_1 = [1,2,0,4.7,'R1']
      2
      3 def invert_lst(lst_1):
      4     for num in lst_1:
      5         try:
      6             print(1/num)
      7         except ZeroDivisionError:
      8             print('erro: divisão por zero!')
      9         except TypeError:
     10             print('erro: não é um número!')
     11
     12 print(invert_lst(lst_1))
     13
```

```
1.0
0.5
erro: divisão por zero!
0.2127659574468085
erro: não é um número!
```

Nesse *script*, na primeira linha, criamos uma lista com 5 elementos, sendo o último deles uma *string*. Depois criamos a função *invert_list(),* que recebe uma lista e inverte cada elemento nela contido, ou seja, divide 1 pelo valor de cada item da lista. Na linha 12 queremos ver o que acontece quando chamamos a função e passamos a lista criada no início para ela. Incluímos o nosso já conhecido par *try/except* dentro da função para detectar algumas possíveis exceções.

Não tivemos problemas na divisão com os dois primeiros itens, que são números inteiros reais. Já o terceiro item, que é 0, não passou no teste de divisão e, por isso, uma mensagem de *ZeroDivisionError* foi enviada ao usuário. O quarto item, um número fracionário, também pôde ser invertido normalmente. Mas o quinto item também causou um erro, agora um *TypeError*, divisão de um inteiro por uma *string*, quando da execução do *script*. Pensamos que agora fechamos de vez esse tema tão importante que é o tratamento de erros e exceções.

Leitura de Arquivos Texto no Python

Os *scripts* (programas) dos experimentos com Python que escrevemos até agora na plataforma *Colab* foram todos armazenados num computador remoto da *Google* na forma de arquivos, que acessamos via *web*. Também podemos criar esses arquivos com editores de textos simples, como o *Notepad*, e salvá-los localmente, no disco do nosso PC.

Todos esses arquivos têm seus próprios nomes, que são únicos para um mesmo projeto. Eles têm também, ligada ao nome, uma extensão identificadora do tipo de arquivo. Os *scripts* escritos na linguagem Python tem a extensão *.py* depois do nome. Os arquivos texto sem formatação tem a extensão *.txt* e arquivos *Word*, que são formatados, vem com *.docx* depois do nome. Até aqui tudo bem.

Um arquivo *.txt* é composto por um bloco com caracteres não formatados estruturados em linhas, cuja abertura não depende de um programa específico. Todos os sistemas operacionais já vem com um editor de textos na forma de um bloco de notas que abre qualquer arquivo *.txt*.

Já os processadores de texto, como o *Word* do *Windows* ou o *Write* do pacote *LibreOffice*, são outra classe de editores, onde podemos formatar títulos com fontes grandes, sublinhar ou colorir partes do texto e até inserir imagens.

Um arquivo formatado somente poderá ser 'entendido' por um programa compatível com aquele que o formatou. Tente abrir um arquivo *.docx* num bloco de notas como o *Notepad* e observe a confusão de caracteres que aparece na tela.

Atualmente tudo e todos estamos em arquivos. Dados, imagens, áudios e vídeos coletados mundo a fora estão disponíveis em arquivos de graça ou pagos na *internet*. Um arquivo é somente uma coleção de símbolos na forma de caracteres *ASCII* (*American Standard Code for Information Interchange*) que codificam uma informação útil e que, assim, pode ser transmitida a qualquer lugar no espaço e no tempo.

Um arquivo *.txt* tanto pode conter um texto simples com linhas de instruções, ou um conjunto de informações coletadas por um sensor distante. Também podemos fazer uma 'raspagem' (em inglês *scraping*) diretamente em uma página na *internet* para coletar alguma informação com o clássico trio: *marcar-copiar-colar,* ou baixar o arquivo, se este estiver disponível para *download*.

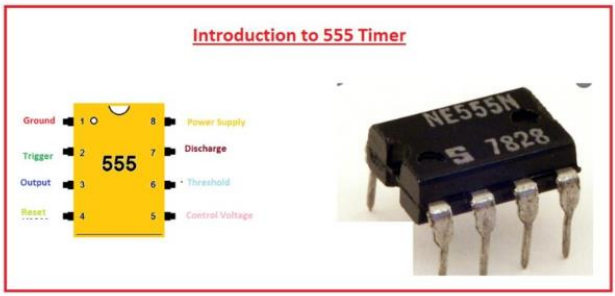
Nos nossos experimentos com Python com leitura e escrita de arquivos do tipo texto, com extensão *.txt*, vamos recolher alguns dados sobre o aniversariante *LM-555*, que neste ano está fazendo 50 anos, da página *web* https://www.theengineeringknowledge.com/introduction-to-555-timer-working-circuit-pinout-applications/.

Abra essa página na *internet* e marque com o botão esquerdo do *mouse* somente o texto do tópico '*Introduction to 555 Timer*'.

Veja a tela a seguir:

Introduction to 555 Timer

- The 555 timer is integrated circuitry used in such applications where usage of pulse, delay of time and oscillation is needed.
- There are 2 modules of 556 in single packaging and four 558 modules in a single casing.
- The first time it was used in 1972. There are many manufacturers are creating this device.
- There are normally twenty-five transistors fifteen resistances and two diodes are positioned a single board.
- This board comes up with eight pinouts that are dual inline packaging.
- Some modules of these categories like 556 come with fourteen pinouts and 558 has sixteen pinouts.
- Its temperature range where it works is between zero to seventy centigrade.
- The voltage range for which it operates is 4.5 volts to fifteen volts dc.
- There are flip flops, voltage dividers and comparators are configured at the board of this device.
- normally it employed to offer the time delay where it used.
- It operation modes are monostable, bistable.
- It comprises of 3 resistances of five kilo-ohms in series combination to voltage dividers modules so-known as 555 timers.
- It provides the larger value current at the output terminal so used to operates TTL.



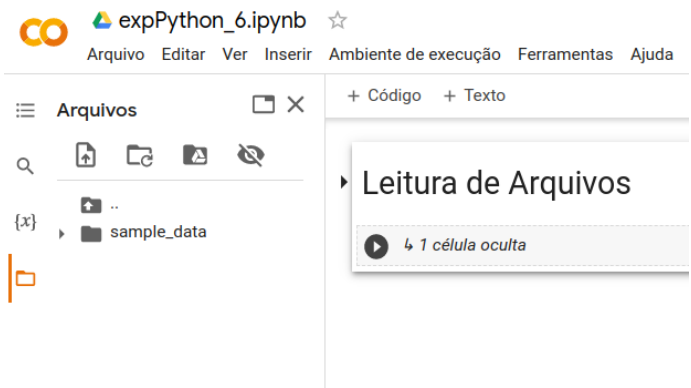
Agora, copie essa região marcada para a área de transferência (*Ctrl+C*) e o cole (*Ctrl+V*) no bloco de notas do seu sistema operacional. Veja que capturamos o texto sem nenhuma formatação diretamente do código *HTML,* que foi enviado por um servidor *web* remoto para o seu navegador de *internet*. Este sim, como os processadores de texto, vai interpretar a formatação *HTML* embutida com o texto e renderizar uma tela no seu monitor. Por fim, salve esse arquivo como, por exemplo, *timer555.txt,* na área de trabalho (*Desktop*) do seu PC.

Lendo um arquivo no disco do PC

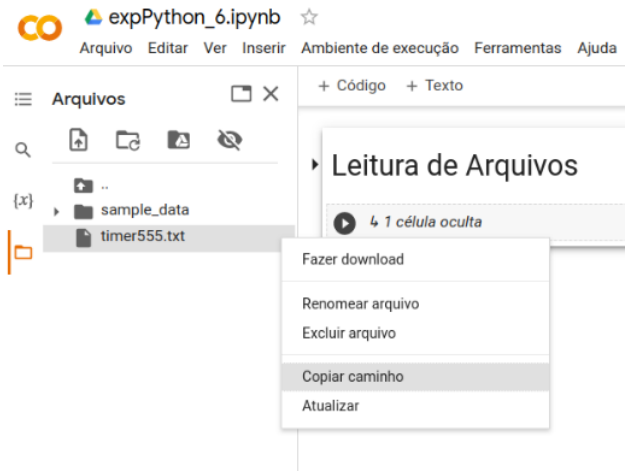
No Python podemos abrir qualquer arquivo *.txt* com o método *open()* de sua biblioteca básica e depois manipular o seu conteúdo. Uma vez aberto, este arquivo passa a ser mais um objeto que se integra aos outros objetos Python criados no *script*. Depois de utilizar o arquivo, devemos sempre fechá-lo com o método *close()*. Vamos abrir e ler o conteúdo do arquivo *timer555.txt* que criamos acima na nossa área de trabalho.

Vejamos primeiro o método *open()* de abertura de arquivos em Python. Esse método, como uma função com retorno em Python, retorna um objeto *manipulador de arquivos* (*file handler*) com o qual podemos realizar diversas operações sobre o conteúdo de um arquivo. Todas as ferramentas que podem afetar o arquivo são chamadas a partir desse objeto *file handler*. Normalmente nomeamos uma variável e lhe atribuímos esse objeto manipulador.

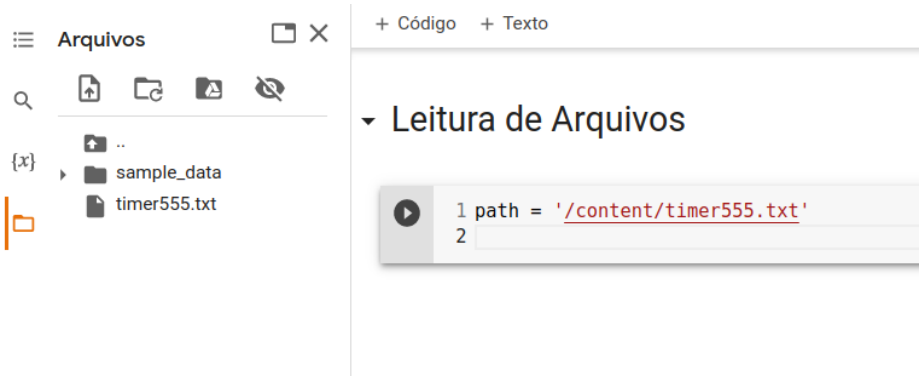
Na sua plataforma *Google Colab* abra à esquerda da tela a aba '*Arquivos*', clicando no ícone como uma pasta; o último à esquerda. Veja a tela a seguir.



Agora clique no ícone com uma seta de *upload* para carregar temporariamente o arquivo no seu ambiente atual de execução. Encontre o arquivo na área de trabalho do seu PC e clique duas vezes nele (ou clique uma vez nele e no botão '*open*'). Copie para a área de transferência o caminho do arquivo, clicando nos 3 pontos ao lado do nome do arquivo e em '*Copiar caminho*'. Como na tela a seguir.



Agora, na primeira célula do *Colab*, crie uma variável chamada *path* e lhe atribua o conteúdo da área de transferência entre aspas simples. Veja a próxima tela.



Para fechar a aba de arquivos, clique no 'X' no topo da aba. OK, agora vamos abrir esse arquivo de texto com o método *open()* no modo *somente leitura*, e atribuir o seu conteúdo à uma variável chamada *file*.

A sintaxe desse método para leitura é: *open('nome_arquivo', 'r')*. Na verdade o segundo parâmetro, *'r'* de *read*, pode até ser dispensado, pois leitura é o modo *default* do método *open()*. A linha 3 do nosso *script* mostra o que temos nesse arquivo *timer555.txt* e a linha 5 fecha o arquivo. Veja a tela *Colab* seguinte.

```
[5]  1 path = '/content/timer555.txt'
     2
     3 file = open(path, 'r')
     4 print(file.read())
     5 file.close()
     6
```

```
Introduction to 555 Timer
The 555 timer is integrated circuitry used in such applications where usage c
There are 2 modules of 556 in single packaging and four 558 modules in a sing
The first time it was used in 1972. There are many manufacturers are creating
There are normally twenty-five transistors fifteen resistances and two diodes
This board comes up with eight pinouts that are dual inline packaging.
Some modules of these categories like 556 come with fourteen pinouts and 558
Its temperature range where it works is between zero to seventy centigrade.
The voltage range for which it operates is 4.5 volts to fifteen volts dc.
There are flip flops, voltage dividers and comparators are configured at the
normally it employed to offer the time delay where it used.
It operation modes are monostable, bistable.
It comprises of 3 resistances of five kilo-ohms in series combination to volt
It provides the larger value current at the output terminal so used to operat
```

Veja que temos 14 linhas com caracteres que formam palavras e estas, frases inteiras. Cada linha termina com um caractere de controle que não vemos: o caractere *LF* (*line-Feed*), que sinaliza o fim da presente linha; e que uma nova linha já pode ser acrescentada ao texto.

Esse comando é herança do conjunto de controles para impressoras matriciais. Seu código na tabela *ASCII* é o decimal 10. No Python o comando *LF* é representado por '\n'. Assim, no final de cada linha do nosso arquivo *timer555.txt* temos também o caractere invisível '\n'.

Se quisermos ler somente a primeira linha do arquivo podemos ou passar o número de caracteres da linha (25 *bytes*) com *print(file.read(25))*, ou com *print(file.readline()).* A ferramenta *readline()* retorna uma nova linha de texto de acordo com um ponteiro que é incrementado a cada nova chamada.

```
[17]  1 path = '/content/timer555.txt'
      2
      3 file = open(path, 'r')
      4 #print(file.read(25))
      5 #print(file.readline())
      6
      7 lst_1 = file.readlines()
      8 print(lst_1[0:2])
      9
     10 file.close()
     11
```

```
['Introduction to 555 Timer\n', 'The 555 timer is integrated circui
```

O método *readlines()* - repare o 's'- retorna cada linha como um elemento de uma lista. No exemplo, queremos ver somente as 3 primeiras linhas do arquivo. Repare também que agora, na lista, o caractere de controle '\n' aparece no final de cada linha.

Nossa variável *file*, na linha 3, o manipulador de arquivos, é um objeto iterável e portanto podemos usá-lo num laço de repetição *for* para listar todas as linhas do arquivo, como no exemplo da tela a seguir.

O método *rstrip()* aplicado a cada elemento iterado serve para eliminar caractere '\n' extra que a função *print()* sempre insere no final de cada linha que mostra na tela.

```
[21]  1 path = '/content/timer555.txt'
      2
      3 file = open(path, 'r')
      4 for i in file:
      5     print(i.rstrip())
      6
      7 file.close()
```

```
Introduction to 555 Timer
The 555 timer is integrated circuitry used in such applications where usage c
There are 2 modules of 556 in single packaging and four 558 modules in a sing
The first time it was used in 1972. There are many manufacturers are creating
There are normally twenty-five transistors fifteen resistances and two diodes
This board comes up with eight pinouts that are dual inline packaging.
Some modules of these categories like 556 come with fourteen pinouts and 558
Its temperature range where it works is between zero to seventy centigrade.
The voltage range for which it operates is 4.5 volts to fifteen volts dc.
There are flip flops, voltage dividers and comparators are configured at the
normally it employed to offer the time delay where it used.
It operation modes are monostable, bistable.
It comprises of 3 resistances of five kilo-ohms in series combination to volt
It provides the larger value current at the output terminal so used to operat
```

Note que não devemos esquecer de fechar o arquivo aberto com o método *close()* depois de usá-lo. Mas, no Python, podemos fechar automaticamente um arquivo usando a declaração *with* antes de abri-lo com o método *open()*, não sendo necessária a invocação do método *close()*. Veja a tela a seguir.

```
[24]  1 path = '/content/timer555.txt'
      2
      3 with open(path, 'r') as file:
      4     print(file.readlines()[0:2])
      5
```

```
['Introduction to 555 Timer\n', 'The 555 timer is integrated circuit
```

Caros leitores, nos estendemos mais do que gostaríamos nesses exemplos de abertura e leitura de arquivos textos com Python. Vamos deixar para os próximos artigos a escrita em arquivos desse tipo, e também a leitura e escrita em arquivos com extensão *.csv* e *xlsx*, com *scripts* Python.

Até breve!

Links:
Livro digital do autor '*Experimentos com o Arduino*': https://www.amazon.com.br
Sobre o Timer 555: https://www.theengineeringknowledge.com/introduction-to-555-timer-working-circuit-pinout-applications/
Sobre a tabela ASCII: https://documentacao.senior.com.br/tecnologia/6.2.35/informacoes-tecnicas/tabela-ascii.htm

João Alexandre Silveira*

No artigo anterior dessa nossa série tratamos da leitura e da escrita de *arquivos texto* não formatados, onde dissemos que um arquivo desse tipo tem a extensão *.txt* e que era composto por um bloco de caracteres *ASCII* organizados em linhas. Vimos que qualquer *editor de textos* simples, que normalmente já vem instalado em todo sistema operacional, pode abrir um arquivo *.txt*, como os blocos de notas do *Windows*, do *Mac* ou do *Linux*.

Também dissemos que, diferentemente dos editores, com os *processadores de texto*, como o *Word* do *Windows* ou o *Write* do *LibreOffice*, podemos formatar partes de um texto; como criar títulos com fontes grandes; sublinhar e colorir palavras e frases; além de inserir imagens em qualquer lugar dentro do arquivo. Um arquivo de texto dessa forma formatado somente poderá ser corretamente aberto por um programa compatível com aquele que o formatou.

Arquivos podem conter dados coletados de sensores, imagens capturadas por câmeras, áudios e vídeos de movimentos nas ruas e prédios; que podem ser compartilhados entre computadores conectados entre si em redes locais ou mundiais, como a *internet*. Tudo à nossa volta e também todos nós estamos em arquivos. Naquele artigo do mês passado fizemos experimentos com um arquivo *.txt* com um texto, que 'raspamos' (*scraping*) de uma página *web*, sobre o clássico circuito integrado LM-555.

*__Autor do livro "Experimentos com o Arduino", disponível em__ www.amazon.com.br

Para abrir esse arquivo criado usamos o método *open()* da biblioteca básica da linguagem Python. Também nesse artigo falamos rapidamente sobre o que são *objetos*, *classes, instâncias, atributos e métodos* em Python; sem nos aprofundarmos em o que é programação orientada a objetos (*OOP – Object-Oriented Programming*).

Nessa sétima etapa de nossa jornada vamos ver como abrir e manipular outros tipos de arquivos um outro tipo de arquivo com *scripts* em Python: os arquivos texto com extensões *csv*.

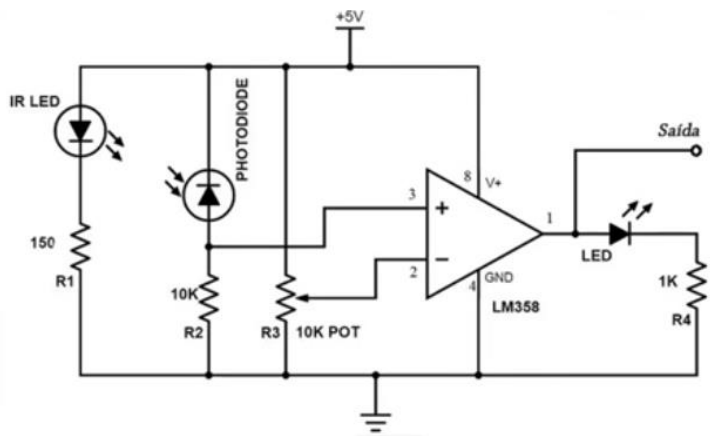
Abrindo Arquivos CSV com Python

Um arquivo *.csv* (*Comma Separated Values*) é composto por um texto simples não formatado, onde cada linha representa um registro de valores separados por vírgulas; é a forma mais simples de se guardar dados em tabelas. Como planilhas, arquivos *.csv* podem ser abertos por programas como o *Excel*, o *LibreOffice Calc* ou o *Google Spreadsheet*.

Vamos então criar um arquivo *csv* simples com o bloco de notas do sistema operacional instalado no nosso PC, e armazená-lo numa pasta na área de trabalho. Depois, vamos ler e manipular as informações contidas nesse arquivo com umas poucas linhas escritas em Python.

Comecemos montando uma tabela com 9 linhas e 3 colunas com os seguintes dados do circuito de um tacômetro simples com o *opamp* LM-358, que tomamos emprestado no seguinte canal *Youtube*: https://www.youtube.com/watch?v=QJvJVSDGTxQ

Veja o circuito e a tabela mostrados abaixo.



ID, Descrição, Valor
D1, IR LED, TIL-32
D2, Fotodiodo, TIL-78
R1, Resistor, 150
R2, Resistor, 10K
P1, Potenciômetro, 10K
CI1, Op Amp, LM-358
D3, LED, 5mm
R4, Resistor, 1K

Conforme descrição no canal do autor, nesse circuito o LED infravermelho está continuamente emitindo um feixe de luz que é refletido por uma marca branca num eixo mecânico giratório para o fotodiodo TIL-78. O *opamp* LM-358 montado como comparador de tensão detecta toda vez que essa marca branca interrompe o feixe de luz e, assim, muda sua saída de nível baixo (terra) para alto (+5 volts). Essa mudança de estado vai fazer o LED verde D3 piscar e ativar a entrada de um frequencímetro ou de um osciloscópio a cada giro completo do eixo.

Copie toda a tabela acima para um bloco de notas e após a linha 9, faça isso!, pressione mais uma vez a tecla *Enter*. Salve o arquivo como, por exemplo, *tacometro.csv* numa pasta na área de trabalho (*Desktop*) do seu PC. Depois abra esse arquivo com um programa de planilha eletrônica e você terá alguma coisa como mostra a figura abaixo.

	A	B	C
1	ID	Descrição	Valor
2	D1	IR LED	TIL-32
3	D2	Fotodiodo	TIL-78
4	R1	Resistor	150
5	R2	Resistor	10K
6	P1	Potenciometro	10K
7	CI1	OpAmp	LM-358
8	D3	LED	5mm
9	R4	Resistor	1K
10			

OK, agora copie o *script* Python abaixo para o seu ambiente *Google Colab* e clique no ícone '*Arquivos*' à esquerda, para abrir uma aba lateral. Nessa aba, clique no ícone com uma seta para cima para fazer o carregamento (*upload*) do arquivo *tacometro.csv* que você guardou numa pasta na área de trabalho do seu PC. Por fim, nos três pontos à direita do arquivo já carregado, clique em '*Copiar caminho*' e cole (*Cntl+V*) dentro dos parênteses da função *open()* na linha 3. Todo esse procedimento já vimos anteriormente na parte 4 desta nossa série.

```
[15]  1 import csv
      2
      3 f = open('/content/tacometro.csv')
      4 csv_tac = csv.reader(f)
      5
      6 for line in csv_tac:
      7     print(line)
      8
```

```
['ID', ' Descrição', ' Valor']
['D1', ' IR LED', ' TIL-32']
['D2', ' Fotodiodo', ' TIL-78']
['R1', ' Resistor', ' 150']
['R2', ' Resistor', ' 10K']
['P1', ' Potenciometro', ' 10K']
['CI1', ' OpAmp', ' LM-358']
['D3', ' LED', ' 5mm']
['R4', ' Resistor', ' 1K']
[]
```

Nesse *script*, na primeira linha importamos a biblioteca *csv* e na linha 3 abrimos com a função *open()* o arquivo *tacometro.csv* que carregamos e atribuímos todo o seu conteúdo à variável *f*. Na linha 4, com a variável *f* criamos o objeto iterável *csv_tac* usando o método *reader()* da biblioteca *csv.* E nas duas linhas seguintes mandamos imprimir cada elemento de *csv_tac* com um laço *for*. Note que cada linha do arquivo *tacometro.csv* agora é uma lista Python. Assim, podemos por exemplo acessar todos os primeiros elementos de cada coluna desse arquivo. Vamos relembrar como isso funciona modificando a linha 7 do nosso *script*, como mostrado na tela *Colab* abaixo.

```
  1 import csv
  2
  3 f = open('/content/tacometro.csv')
  4 csv_tac = csv.reader(f)
  5
  6 for line in csv_tac:
  7     print(line[0])
  8
```

```
ID
D1
D2
R1
R2
P1
CI1
D3
R4
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-16-1221c59bcf26> in <module>()
      5
      6 for line in csv_tac:
----> 7     print(line[0])
      8

IndexError: list index out of range
```

Ôpa!, apareceu um *erro*! Na verdade uma *exceção*, como já aprendemos nas partes V e VI dessa série. O interpretador Python nos diz que na linha 7 ocorreu uma tentativa de acessar um elemento inexistente numa lista. Como assim, se todos os primeiros elementos de cada coluna foram mostrados corretamente?!

Será que o leitor que acompanhou até aqui toda essa série de artigos saberia dizer onde está esse erro? Uma dica: observe cada linha impressa pelo *script* da página anterior. Aquele *script* imprimiu uma lista em cada uma das 10 linhas; mas, observe a última linha.

Isso mesmo: a última lista está vazia, não existe nenhum elemento nela para ser acessado, daí o erro. E como essa lista foi criada? Ela foi criada depois da linha 9 quando pressionamos a tecla *Enter* e criamos mais uma linha, essa vazia.

Para corrigir essa exceção, abra de novo o arquivo *tacometro.csv*, posicione o cursor nessa última linha, pressione a tecla *Return* e salve novamente o arquivo. Agora abra novamente a aba esquerda no *Colab*, refaça o carregamento do arquivo e execute o *script*. Veja a tela a seguir, onde agora mostramos as duas primeiras colunas de cada linha do arquivo *tacometro.csv*; não existe mais nenhuma exceção quando executamos o *script*!

Lembre-se que devemos sempre fechar, com a função *close()*, na linha 9, um arquivo aberto depois de usá-lo.

```
[21]  1 import csv
      2
      3 f = open('/content/tacometro.csv')
      4 csv_tac = csv.reader(f)
      5
      6 for line in csv_tac:
      7     print(line[0:2])
      8
      9 f.close()
```

```
['ID', ' Descrição']
['D1', ' IR LED']
['D2', ' Fotodiodo']
['R1', ' Resistor']
['R2', ' Resistor']
['P1', ' Potenciometro']
['CI1', ' OpAmp']
['D3', ' LED']
['R4', ' Resistor']
```

Podemos também fechar automaticamente o arquivo depois de usá-lo, se empregarmos a declaração *with* como na listagem abaixo. Observe que incluímos na linha 3, depois do nome do arquivo, a opção '*r*' (*read*). Se nenhum modo for implicitamente indicado dentro da função *open()*, o modo leitura é selecionado por *default*.

```
[20]  1 import csv
      2
      3 with open('/content/tacometro.csv', 'r') as f:
      4     csv_tac = csv.reader(f)
      5     for line in csv_tac:
      6         print(line)
```

```
['ID', ' Descrição', ' Valor']
['D1', ' IR LED', ' TIL-32']
['D2', ' Fotodiodo', ' TIL-78']
['R1', ' Resistor', ' 150']
['R2', ' Resistor', ' 10K']
['P1', ' Potenciometro', ' 10K']
['CI1', ' OpAmp', ' LM-358']
['D3', ' LED', ' 5mm']
['R4', ' Resistor', ' 1K']
```

Mas, vamos em frente. A primeira linha do nosso arquivo *tacometro.csv* é o cabeçalho das colunas com as informações sobre os componentes do circuito que propomos. Podemos separar esse cabeçalho do resto do nosso arquivo usando a função *enumerate()*, que já vimos em artigos anteriores, da seguinte forma:

```
[22]  1 import csv
      2
      3 with open('/content/tacometro.csv') as f:
      4     csv_tac = csv.reader(f)
      5     for line_0, line_ in enumerate(csv_tac):
      6         if line_0 == 0:
      7             print('Cabeçalho:')
      8             print(line_)
      9             print('Dados:')
     10         else:
     11             print(line_)
```

```
Cabeçalho:
['ID', ' Descrição', ' Valor']
Dados:
['D1', ' IR LED', ' TIL-32']
['D2', ' Fotodiodo', ' TIL-78']
['R1', ' Resistor', ' 150']
['R2', ' Resistor', ' 10K']
['P1', ' Potenciometro', ' 10K']
['CI1', ' OpAmp', ' LM-358']
['D3', ' LED', ' 5mm']
['R4', ' Resistor', ' 1K']
```

Uma outra forma, mais simples, de eliminar o cabeçalho é usando a função *next()*, como na tela a seguir:

```
[21]  1 import csv
      2 with open('/content/tacometro.csv') as f:
      3     csv_tac = csv.reader(f)
      4
      5     next(csv_tac)           # salta a primeira linha
      6     for line in csv_tac:    # mostra somente os dados
      7         print(line)
```

```
['D1', ' IR LED', ' TIL-32']
['D2', ' Fotodiodo', ' TIL-78']
['R1', ' Resistor', ' 150']
['R2', ' Resistor', ' 10K']
['P1', ' Potenciometro', ' 10K']
['CI1', ' OpAmp', ' LM-358']
['D3', ' LED', ' 5mm']
['R4', ' Resistor', ' 1K']
```

Escrevendo em Arquivos CSV com Python

Agora vamos acrescentar mais uma coluna ao nosso arquivo, uma lista com o preço de cada componente no circuito. Veja abaixo a lista *P* e o *script*.

P = [1.0, 1.2, 0.2, 0.2, 3.5, 4.5, 1.5, 0.2]

```
[72]  1 import csv
      2 # lista de preços dos componentes:
      3 P = [1.0, 1.2, 0.2, 0.2, 3.5, 4.5, 1.5, 0.2]
      4
      5 with open('/content/tacometro.csv') as f_1:
      6     with open('tacometro_2.csv', 'w') as f_2:
      7         writer = csv.writer(f_2)
      8
      9         for i,line in enumerate(csv.reader(f_1)):
     10             if line[0] == 'ID':
     11                 writer.writerow(line+['Preço'])
     12             else:
     13                 writer.writerow(line+[P[i-1]])
     14
```

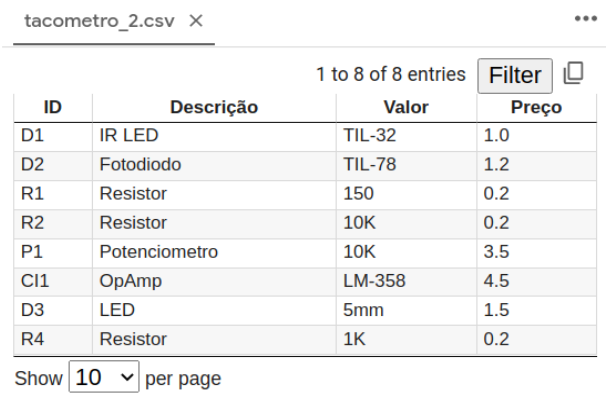
Nesse *script* abrimos normalmente para leitura o arquivo original, *tacometro.csv*, com a função *open()*. Depois, na linha 6, também com a função *open()*, criamos um novo arquivo, *tacometro_2.csv*, no modo escrita com a opção '*w*' (*write*). Na linha 7 criamos o objeto *writer* para esse novo arquivo com o método *writer()* da biblioteca *csv*, previamente importada. Na linha 9 vamos lendo cada linha do primeiro arquivo com o método *reader()* dessa biblioteca; também criamos um cabeçalho para a nova coluna, na linha 11, com a função *writerow()*; por fim, vamos tomando cada elemento da lista *P* e, também com a função *writerow()*, o acrescentando ao novo arquivo, na linha 13.

Vejamos agora na forma de listas, as 8 linhas desse novo arquivo.

```
[85]  1 import csv
      2 with open('/content/tacometro_2.csv') as f:
      3     csv_tac = csv.reader(f)
      4
      5     next(csv_tac)           # salta a primeira linha
      6     for line in csv_tac:    # mostra somente os dados
      7         print(line)
```

```
['D1', ' IR LED', ' TIL-32', '1.0']
['D2', ' Fotodiodo', ' TIL-78', '1.2']
['R1', ' Resistor', ' 150', '0.2']
['R2', ' Resistor', ' 10K', '0.2']
['P1', ' Potenciometro', ' 10K', '3.5']
['CI1', ' OpAmp', ' LM-358', '4.5']
['D3', ' LED', ' 5mm', '1.5']
['R4', ' Resistor', ' 1K', '0.2']
```

Abra a aba à esquerda do ambiente *Colab* e dê um duplo clique sobre esse novo arquivo *tacometro_2.csv*. À direita da tela deverá aparecer na forma de planilha todo o seu conteúdo. Veja a tela abaixo.


tacometro_2.csv

1 to 8 of 8 entries Filter

ID	Descrição	Valor	Preço
D1	IR LED	TIL-32	1.0
D2	Fotodiodo	TIL-78	1.2
R1	Resistor	150	0.2
R2	Resistor	10K	0.2
P1	Potenciometro	10K	3.5
CI1	OpAmp	LM-358	4.5
D3	LED	5mm	1.5
R4	Resistor	1K	0.2

Show 10 per page

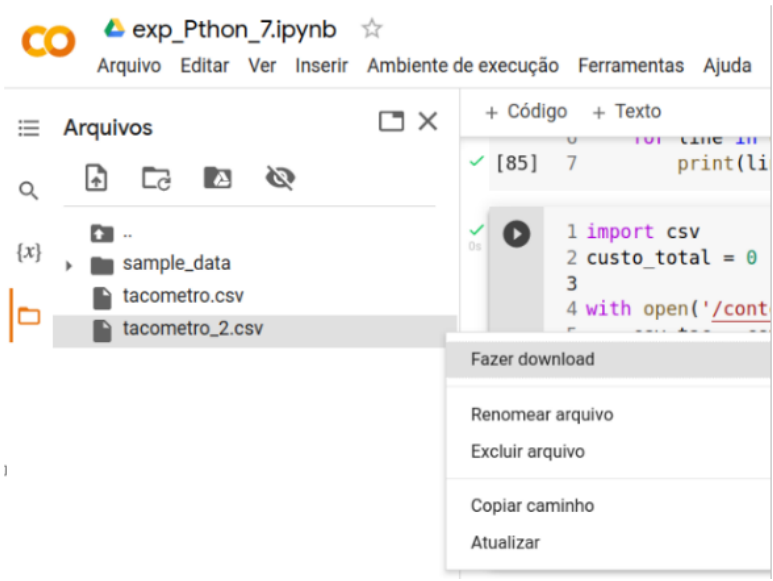
Na quarta coluna desse novo arquivo temos os preços de cada componente do circuito; então podemos calcular o custo total para montar nosso tacômetro da seguinte forma:

```
[112]  1 import csv
       2 custo_total = 0     # atribui 0 a variavel 'custo_total'
       3
       4 with open('/content/tacometro_2.csv') as f:
       5     csv_tac = csv.reader(f)
       6     next(csv_tac)           # salta a primeira linha
       7
       8     for line in csv_tac:
       9         custo_total += float(line[3])
      10
      11 print(f'Custo total do circuito: R$ {custo_total}')
```

```
Custo total do circuito: R$ 12.3
```

Salvando Arquivos CSV com Python

Para salvar esse novo arquivo na mesma pasta onde está o arquivo original, clique nos 3 pontos ao lado do nome do arquivo *tacometro_2.csv* e na opção '*Fazer download*', como mostra a tela abaixo, e selecione a pasta na sua área de trabalho.



No terceiro artigo tivemos os primeiros contatos com a biblioteca *pandas* da linguagem Python; onde vimos o que é uma *Serie* e um *Dataframe*. Agora vamos transformar o arquivo *tacometro_2.csv* num *dataframe* (*dataset*) com o método *DataFrame()* dessa biblioteca *pandas*.

```
[143]  1 import csv
       2 import pandas as pd
       3
       4 with open('/content/tacometro_2.csv') as f:
       5     df_1 = pd.DataFrame(csv.reader(f))
       6
       7 df_1.rename(columns=df_1.iloc[0], inplace = True)
       8 df_1.drop(df_1.index[0], inplace = True)
       9
      10 print(df_1)
      11
      12 df_1.to_csv('df_1.csv')
      13
```

```
    ID       Descrição    Valor Preço
1   D1          IR LED   TIL-32   1.0
2   D2       Fotodiodo   TIL-78   1.2
3   R1        Resistor      150   0.2
4   R2        Resistor      10K   0.2
5   P1   Potenciometro      10K   3.5
6  CI1           OpAmp   LM-358   4.5
7   D3             LED      5mm   1.5
8   R4        Resistor       1K   0.2
```

Nesse *script* nas duas primeiras linhas importamos as bibliotecas *csv* e *pandas.* Depois convertemos o arquivo *tacometro_2.csv* num *dataframe*, *df_1;* e fizemos, na linha 7, da primeira linha do arquivo, o cabeçalho, o nome das colunas; antes eram números inteiros. Na linha 8 tivemos que eliminar do *dataframe* essa primeira linha, que apareceria como dados junto com as outras. Na última linha salvamos esse *dataframe* num outro arquivo csv: *df_1.csv*.

Já dissemos ao longo dessa nossa série que sempre podemos escrever um mesmo programa em Python de diferentes maneiras. Veja como podemos reescrever o *script* acima de uma outra forma mais simplificada, usando o recurso da transposição entre linhas e colunas do *dataframe*; depois setamos a primeira linha como índice e, novamente, fazemos a transposição do *dataframe*; tudo isso numa única linha de comando; como na tela abaixo.

```
[147]  1 import csv
       2 import pandas as pd
       3
       4 with open('/content/tacometro_2.csv') as f:
       5     df_1 = pd.DataFrame(csv.reader(f))
       6
       7 df_1 = df_1.T.set_index(0).T
       8
       9 print(df_1)
```

```
0   ID       Descrição    Valor Preço
1   D1          IR LED   TIL-32   1.0
2   D2       Fotodiodo   TIL-78   1.2
3   R1        Resistor      150   0.2
4   R2        Resistor      10K   0.2
5   P1   Potenciometro      10K   3.5
6  CI1           OpAmp   LM-358   4.5
7   D3             LED      5mm   1.5
8   R4        Resistor       1K   0.2
```

Vamos fechar essa sétima parte do nosso '*Experimentos com a Linguagem Python para Técnicos em Eletrônica*', manipulando nosso arquivo *csv*, agora com a biblioteca *pandas*.

Começamos importando essa biblioteca para nosso *script* e atribuindo o método de leitura de arquivos *read_csv()* a variável *csv_tac*. Uma vantagem do método *read_csv()* é que por *default* ele já utiliza a primeira linha de um arquivo *csv* como cabeçalho das colunas, criando assim um dataframe pronto. Veja a tela a seguir.

```
[153]  1 import pandas as pd
       2
       3 csv_tac = pd.read_csv('/content/tacometro_2.csv')
       4 print(csv_tac)
```

```
    ID       Descrição    Valor  Preço
0   D1          IR LED   TIL-32    1.0
1   D2       Fotodiodo   TIL-78    1.2
2   R1        Resistor      150    0.2
3   R2        Resistor      10K    0.2
4   P1   Potenciometro      10K    3.5
5  CI1           OpAmp   LM-358    4.5
6   D3             LED      5mm    1.5
7   R4        Resistor       1K    0.2
```

Uma vez criado o *dataframe*, podemos realizar diversas manipulações em sua estrutura; como inserir ou eliminar linhas ou colunas; fazer operações matemáticas entre suas células e exportar as modificações feitas.

O método *to_csv()* do *pandas* salva o *dataframe* como arquivo *csv*, como já vimos na página anterior. Veja abaixo como fica o *dataframe* quando fazemos a transposição entre linhas e colunas.

```
[160]  1 import pandas as pd
       2
       3 csv_tac = pd.read_csv('/content/tacometro_2.csv')
       4 csv_tac = csv_tac.T
       5
       6 print(csv_tac)
```

```
                    0           1          2          3               4  \
ID                 D1          D2         R1         R2              P1
 Descrição    IR LED   Fotodiodo   Resistor   Resistor   Potenciometro
 Valor        TIL-32      TIL-78        150        10K             10K
Preço            1.0         1.2        0.2        0.2             3.5

                    5     6          7
ID                 CI1    D3         R4
 Descrição     OpAmp   LED   Resistor
 Valor        LM-358   5mm         1K
Preço            4.5   1.5        0.2
```

No próximo artigo continuaremos ainda tratando da leitura e escrita de arquivos em Python, mas dessa vez de arquivos com extensão *.xlsx*, utilizados em planilhas eletrônicas como o *Excel* da *Microsoft*.

Até lá!

João Alexandre Silveira*

Nesta parte VIII desta nossa série de artigos sobre a linguagem Python para Técnicos e Engenheiros em Eletrônica, trataremos da leitura e escrita de arquivos no formato *Microsoft Excel*, aqueles arquivos com extensão *xls* e *xlsx*. Esses arquivos são, na verdade, tabelas de valores na forma de *planilhas eletrônicas*; uma matriz de células individuais que podem ser facilmente acessadas por um par de *coordenadas x-y*.

As planilhas eletrônicas surgiram como *folhas de cálculos* com o programa *VisiCalc* na época dos primeiros computadores pessoais dos anos 1980. Como diz a Wikipédia, o VisiCalc foi provavelmente a aplicação que fez com que computadores pessoais deixassem de ser um *hobby* e passassem a ser considerados como uma ferramenta de negócios. Depois vieram o *Lotus123*, que incluía um gerenciador de banco de dados, e o, até hoje, consagrado *Excel* da empresa do Bill Gates III.

Nas partes VI e VII desta série falamos sobre o tratamento com Python de arquivos com extensão *.txt* (texto) e *.csv* (*comma separated values*). Na parte VI vimos que arquivos .txt são uma coleção de símbolos na forma de caracteres ASCII que não têm qualquer formatação, e por isso podem ser criados ou abertos por quaisquer editores de textos simples; como aqueles que já vêm junto com os sistemas operacionais instalados no modernos PCs, como o *Notepad* do *Windows*.

*__Autor do livro "Experimentos com o Arduino", disponível em__ www.amazon.com.br

Arquivos com formatação própria, como os criados por processadores de texto, uma outra classe de editores de texto, como o *Word* da Microsoft com extensão *.doc* ou *.docx*, só podem ser criados ou abertos por programas compatíveis com aquele que criou esse tipo de arquivo.

Um arquivo .txt tanto pode conter um texto simples com linhas de instruções para um processador digital, como também um conjunto de informações coletadas por um sensor distante numa rede *internet das coisas*.

Na parte VII vimos outro tipo de arquivo texto sem formatação, aqueles com extensão *csv*. Arquivos csv armazenam tabelas com valores distintos, mas separados por vírgulas, e que podem ser abertos normalmente por programas de planilhas eletrônicas como o Excel ou o *Spreadsheet* da *Google*.

Mas, para abrir arquivos desse tipo com Python, precisamos importar a biblioteca csv e usamos a função nativa *open()* para tratar o arquivo *tacometro.csv*, que criamos nessa parte VII, a partir da tabela de componentes eletrônicos de um circuito de um tacômetro simples com o opamp LM-358.

Aqui repetimos o *script* Python que lê um arquivo csv e separa o cabeçalho dos dados do arquivo em listas. Tudo em Python que pode ser decomposto em listas é facilmente manipulável.

```
[22]  1 import csv
      2
      3 with open('/content/tacometro.csv') as f:
      4     csv_tac = csv.reader(f)
      5     for line_0, line_ in enumerate(csv_tac):
      6         if line_0 == 0:
      7             print('Cabeçalho:')
      8             print(line_)
      9             print('Dados:')
     10         else:
     11             print(line_)
```

```
Cabeçalho:
['ID', ' Descrição', ' Valor']
Dados:
['D1', ' IR LED', ' TIL-32']
['D2', ' Fotodiodo', ' TIL-78']
['R1', ' Resistor', ' 150']
['R2', ' Resistor', ' 10K']
['P1', ' Potenciometro', ' 10K']
['CI1', ' OpAmp', ' LM-358']
['D3', ' LED', ' 5mm']
['R4', ' Resistor', ' 1K']
```

Ainda na parte VII, criamos o arquivo *tacometro_2.csv* onde incluímos uma coluna com os preços dos componentes do circuito com *opamp*. Nosso arquivo *tacometro_2.csv* pode ser aberto normalmente numa planilha Excel no *Google Colab*, como mostrada na tela abaixo.

tacometro_2.csv ×

1 to 8 of 8 entries Filter

ID	Descrição	Valor	Preço
D1	IR LED	TIL-32	1.0
D2	Fotodiodo	TIL-78	1.2
R1	Resistor	150	0.2
R2	Resistor	10K	0.2
P1	Potenciometro	10K	3.5
CI1	OpAmp	LM-358	4.5
D3	LED	5mm	1.5
R4	Resistor	1K	0.2

Show 10 per page

Nessas duas partes anteriores sobre leitura e escrita de arquivos, usamos a mesma função *open()* da biblioteca básica da linguagem Python. Agora, nesse artigo vamos falar de duas bibliotecas, *pandas* e *Openpyxl*, que devemos importar para poder integrar Python com Excel. Mas, antes, vejamos um painel que subtrai da página web https://realpython.com/openpyxl-excel-spreadsheets-python/ e que resume a terminologia básica de uma planilha eletrônica como o Excel.

Learning Some Basic Excel Terminology

Here's a quick list of basic terms you'll see when you're working with Excel spreadsheets:

Term	Explanation
Spreadsheet or Workbook	A **Spreadsheet** is the main file you are creating or working with.
Worksheet or Sheet	A **Sheet** is used to split different kinds of content within the same spreadsheet. A **Spreadsheet** can have one or more **Sheets**.
Column	A **Column** is a vertical line, and it's represented by an uppercase letter: *A*.
Row	A **Row** is a horizontal line, and it's represented by a number: *1*.
Cell	A **Cell** is a combination of **Column** and **Row**, represented by both an uppercase letter and a number: *A1*.

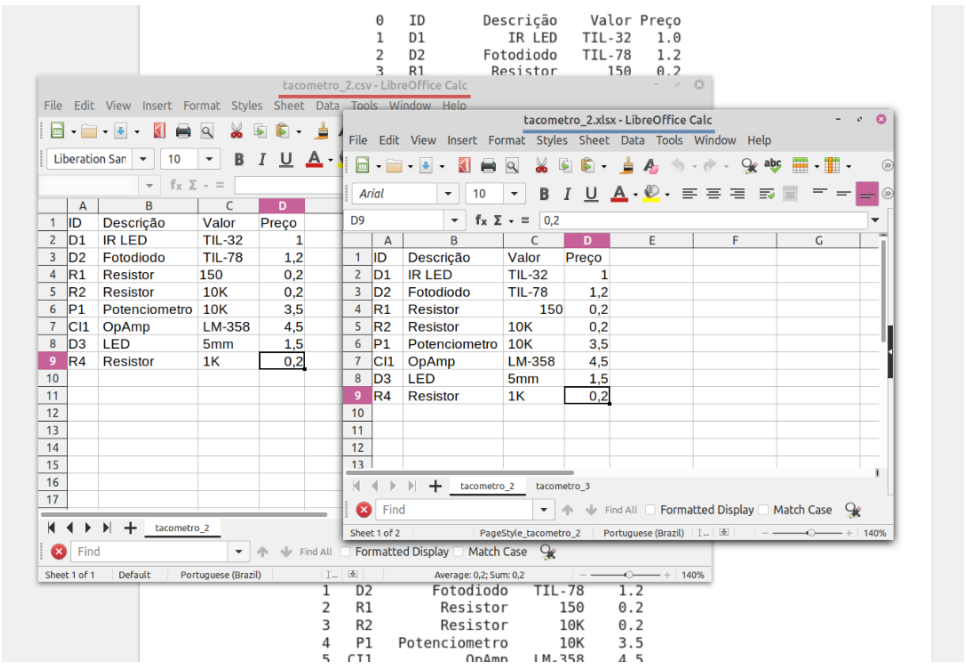
A integração Python – Excel

No Excel e em todas as planilhas eletrônicas existe a função =*SUM()*, com o símbolo grego *sigma*, que adiciona valores de grupos de células tomadas de uma mesma coluna. Já conhecemos uma versão dessa função Excel implementada com Python com poucas linhas de comandos na parte VII dessa nossa série.

```
[112]  1 import csv
       2 custo_total = 0     # atribui 0 a variavel 'custo_total'
       3
       4 with open('/content/tacometro_2.csv') as f:
       5     csv_tac = csv.reader(f)
       6     next(csv_tac)           # salta a primeira linha
       7
       8     for line in csv_tac:
       9         custo_total += float(line[3])
      10
      11 print(f'Custo total do circuito: R$ {custo_total}')
```

```
Custo total do circuito: R$ 12.3
```

O arquivo *tacometro_2.csv* ,criado na parte VII e aberto no Excel, pode ser salvo também como *tacometro_2.xlsx*.



Integração usando pandas

Se importarmos a biblioteca *pandas*, já nossa conhecida, podemos tratar uma planilha Excel como uma base de dados, e com poder de desfazer a estrutura original do arquivo. No *script* abaixo importamos a biblioteca *pandas* como *pd* e atribuímos à variável *taco_df* o arquivo *Excel, tacometro_2.xlsx*. Por *default,* a primeira linha da planilha Excel é considerada pelo *pandas* como cabeçalho e por isso as células dessa linha vão ser os nomes das colunas da planilha.

O módulo *read.excel()* dessa biblioteca transforma uma planilha *Excel* num objeto *dataframe*, uma matriz de dados que podem ser manipulados de diferentes formas. Veja que a função *display()* mostra o conteúdo da planilha de uma forma mais visual que a função *print()*.

```
1 taco_df = pd.read_excel('/content/tacometro_2.xlsx')
2 print(taco_df)
3 print()
4 display(taco_df)
```

```
   ID       Descrição    Valor  Preço
0  D1          IR LED   TIL-32    1.0
1  D2       Fotodiodo   TIL-78    1.2
2  R1        Resistor      150    0.2
3  R2        Resistor      10K    0.2
4  P1   Potenciometro      10K    3.5
5 CI1           OpAmp   LM-358    4.5
6  D3             LED      5mm    1.5
7  R4        Resistor       1K    0.2
```

	ID	Descrição	Valor	Preço
0	D1	IR LED	TIL-32	1.0
1	D2	Fotodiodo	TIL-78	1.2
2	R1	Resistor	150	0.2
3	R2	Resistor	10K	0.2
4	P1	Potenciometro	10K	3.5
5	CI1	OpAmp	LM-358	4.5
6	D3	LED	5mm	1.5
7	R4	Resistor	1K	0.2

Podemos pôr os valores da coluna ‘Preço’ numa lista, ou somente listar na tela toda a coluna usando do *pandas* o parâmetro *usecols = [‘Preços’]* quando da abertura do arquivo. Ou de outra forma, o localizador de posições *pandas* baseado no nome da coluna, o método *loc[]*.

```
[5]  1 print(taco_df['Preço'].tolist())
     2
```

```
[1.0, 1.2, 0.2, 0.2, 3.5, 4.5, 1.5, 0.2]
```

```
[6]  1 taco_df = pd.read_excel('/content/tacometro_2.xlsx', usecols = ['Preço'])
     2 print(taco_df)
```

```
   Preço
0    1.0
1    1.2
2    0.2
3    0.2
4    3.5
5    4.5
6    1.5
7    0.2
```

```
[7]  1 taco_df.loc[:,'Preço']  # todas as celulas (linhas) da coluna Preço
```

```
0    1.0
1    1.2
2    0.2
3    0.2
4    3.5
5    4.5
6    1.5
7    0.2
Name: Preço, dtype: float64
```

Temos agora uma planilha Excel na forma de um dataframe (um *dataset*), e podemos manipular cada célula dessa planilha com comandos Python da biblioteca *pandas*. Nessa tela listamos todas as informações do opamp do circuito na planilha.

```
[85]  1
      2 print(taco_df.loc[5])
```

```
ID                CI1
 Descrição      OpAmp
 Valor         LM-358
Preço             4.5
Name: 5, dtype: object
```

Integração usando Openpyxl

A outra biblioteca mais usada para abrir e modificar planilhas Excel com Python é a *Openpyxl*. Essa biblioteca, como a *pandas*, precisa estar instalada na sua versão de Python. No *Google Colab*, a nossa plataforma de desenvolvimento *on-line* que usamos até aqui em todos os nossos experimentos com a linguagem, instalamos com '*!pip install openpyxl*'.

```
[16]  1 !pip install openpyxl           # instalar biblioteca openpyxl
```

```
[6]   1 import openpyxl                 # importa biblioteca
```

```
[15]  1 # carrega arquivo excel na variavel taco_xls
      2 taco_xls = openpyxl.load_workbook('/content/tacometro_2.xlsx')
      3
      4 sheet = taco_xls['tacometro_2'] # seleciona uma planilha do arquivo
      5 print(sheet.title)              # mostra nome da planilha
      6 print(sheet.dimensions)         # tamanho
      7 print(sheet.active_cell)        # celula ativa (no foco)
```

```
tacometro_2
A1:D9
D9
```

Depois de importar a biblioteca *openpyxl,* carregamos o arquivo *xlsx* na variável *taco_xls* e selecionamos a primeira planilha, como *sheet* (folha). Depois checamos o título da planilha, seus limites (da célula *A1* até *D9*) e qual célula está com foco (*D9* clicada).

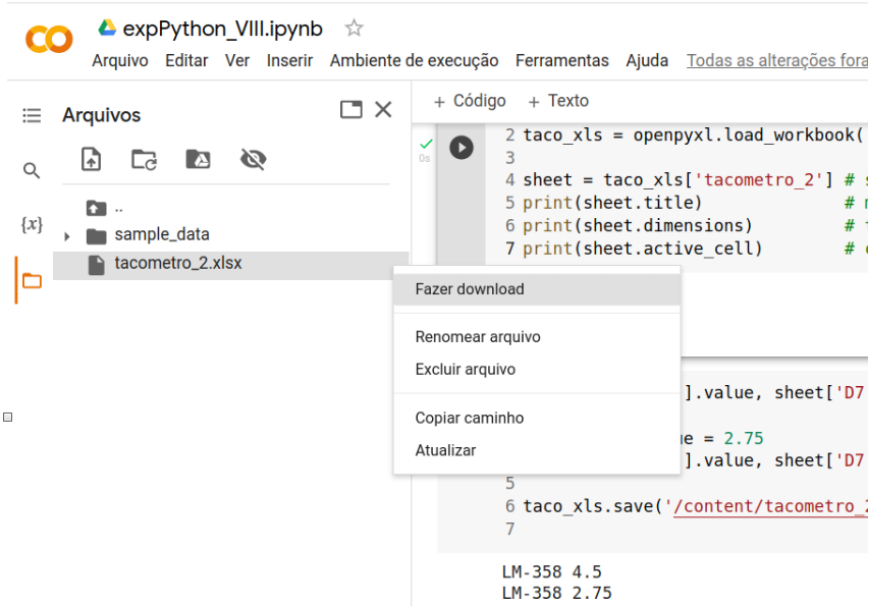
Agora, digamos que queremos mudar o preço do opamp de $4.5 para $2.75 na planilha. Basta indicarmos na planilha a posição x-y da célula cujo valor queremos alterar, junto com a propriedade *value* e o novo valor; depois, devemos salvar a mudança no arquivo original, como vemos na tela abaixo.

```
[29]  1 print(sheet['C7'].value, sheet['D7'].value) # mostra preço atual do opamp
      2
      3 sheet['D7'].value = 2.75                    # atualiza preço do opamp
      4 print(sheet['C7'].value, sheet['D7'].value)
      5
      6 taco_xls.save('/content/tacometro_2.xlsx')  # salva planilha atualizada.
      7
```

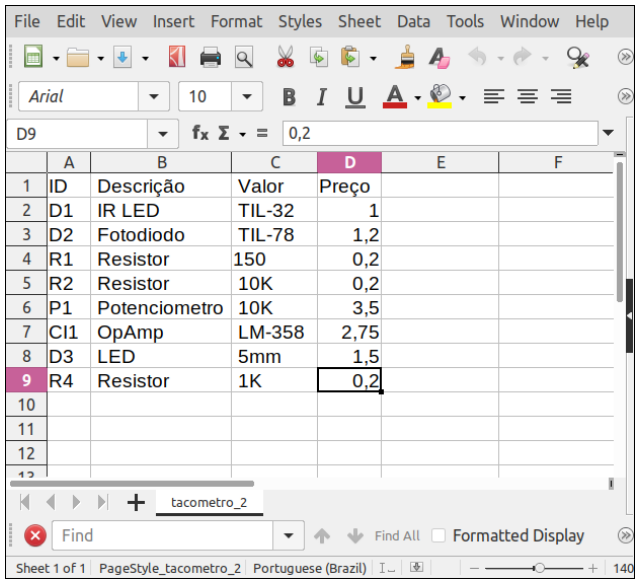
```
LM-358 4.5
LM-358 2.75
```

No *Google Colab,* depois de salvar o arquivo xlsx, será necessário baixar esse arquivo que salvamos e atualizar o original na pasta que criamos na área de trabalho.

É só clicar em nos 3 pontinhos ao lado do arquivo xlsx à esquerda da tela e em ‘Fazer download’; como na tela a seguir.



Depois é só clicar em '*Recarregar*' no menu da planilha aberta.



	A	B	C	D	E	F
1	ID	Descrição	Valor	Preço		
2	D1	IR LED	TIL-32	1		
3	D2	Fotodiodo	TIL-78	1,2		
4	R1	Resistor	150	0,2		
5	R2	Resistor	10K	0,2		
6	P1	Potenciometro	10K	3,5		
7	CI1	OpAmp	LM-358	2,75		
8	D3	LED	5mm	1,5		
9	R4	Resistor	1K	0,2		
10						
11						
12						

Podemos inserir uma nova planilha no arquivo xlsx em que estamos trabalhando usando a propriedade *create_sheet()*, como na tela a seguir. Depois de salvar e descarregar para a pasta de trabalho, atualize o arquivo *tacometro_2.xlsx*.

```
[35]  1 # insere uma nova planilha no arquivo xlsx
      2 taco_xls_1 = taco_xls.create_sheet('tacometro_3')
      3 taco_xls.save('/content/tacometro_2.xlsx')
```

Agora, se queremos incluir numa célula da nossa planilha a data de sua atualização e em outra a formula de somatória de uma coluna, usamos a biblioteca *datetime* e a função *today()* e fazemos como no *script* abaixo.

```
[76]  1 import datetime
      2 from datetime import date
      3
      4 today = date.today()
      5 print('Hoje: ', today)
      6 sheet['A10'].value = today
      7
```

```
Hoje:  2022-07-14
```

```
[83]  1 sheet['A12'].value = '=SUM(D1, D9)'
```

```
[84]  1 print(sheet['A10'].value)
      2 print(sheet['A12'].value)
```

```
2022-07-14
=SUM(D1, D9)
```

Aqui nós importamos a classe '*date*' do módulo importado '*datetime*' e criamos um objeto variável '*today*' para guardar a data local do dia atual, hoje, e a colocamos na célula *A10*. Na célula *A12* colocamos uma fórmula das planilhas eletrônicas: a soma de um grupo de células numa coluna.

Ainda sobre a célula *A10* que aponta para a data local atualizada, podemos configurá-la de diversas formas, como em dois exemplos na tela seguinte.

```
[95]  1 # dd/mm/YY
      2 d1 = today.strftime("%d/%m/%Y")
      3 print('Hoje é dia', today.strftime("%d/%m/%Y"))
      4
      5 # dia - mes abreviado - ano
      6 d4 = today.strftime("%d-%b-%Y")
      7 print('Atualizado em', today.strftime("%d-%b-%Y"))
```

```
Hoje é dia 14/07/2022
Atualizado em 14-Jul-2022
```

Conclusão

Com essas duas ferramentas, as bibliotecas *pandas* e *openpyxl*, podemos, através de comandos simples da linguagem Python, intervir em qualquer célula de uma planilha eletrônica; seja somente lendo seu conteúdo ou mesmo alterando seu valor. Podemos criar novas planilhas dentro de um arquivo *xlsx* já existente e podemos também apagar planilhas ou arquivos inteiros desse tipo.

O assunto é extenso e cabe ao leitor, conhecedor de Excel e curioso por novas possibilidades, buscar conhecer mais a fundo essas duas bibliotecas Python.

Mês que vem nos vemos de novo. Até lá!

João Alexandre Silveira*

Criação de Interfaces Gráficas com o Tkinter

Nas três partes anteriores dessa nossa série tratamos da leitura e escrita de arquivos *txt* (texto), *csv* (*comma separated values*) e *xlsx* (*Microsoft Excel)* com as bibliotecas *csv*, *pandas* e *Openpyxl* da linguagem Python. Vimos que essas bibliotecas contêm funções e métodos com as quais podemos facilmente abrir qualquer um desses arquivos; e, uma vez abertos, podemos modificar quaisquer caracteres ou valores neles contidos e depois salvá-los normalmente, com o mesmo nome ou com um outro nome qualquer.

Até aqui todos os *scripts* ou programas que escreve*mos* em *Python* foram testados na plataforma *Google Colab* ou numa interface de linha de comando conhecida por *Shell*. Avançando, nesta nona parte, chegou a hora de conhecermos mais uma plataforma de criação de *scripts* em Python e também tratarmos da criação de interfaces gráficas entre um programa Python e o usuário desse programa. Para isso vamos testar o editor *VSCode* e conhecer mais uma biblioteca nativa do Python: a biblioteca *Tkinter*.

Mas, antes, o que é uma interface? Esse substantivo feminino vem da palavra inglesa com a mesma grafia que quer dizer ‘entre faces’, ou entre formas diferentes. Na Física é a superfície que forma o limite comum entre dois corpos ou espaços. Na Eletrônica é um circuito que compatibiliza dois outros circuitos diferentes de modo que estes possam trocar sinais elétricos entre si; como um *conversor ADC*, que faz com que a saída de um circuito analógico possa ser conectada a entrada de um circuito digital.

***Autor do livro “Experimentos com o Arduino”, disponível em** www.amazon.com.br

Um painel de um instrumento eletrônico é também uma interface, agora entre as funções do instrumento e o operador deste. No mundo dos computadores uma interface tanto pode ser também um painel físico para acesso direto com um *hardware*, ou um painel virtual, por onde o operador pode interagir com o *software* que é executado naquele *hardware*.

É dessa última definição, a interação via painel virtual, normalmente chamada de *GUI* (*Graphical User Interface*) ou interface gráfica com o usuário, que trataremos hoje em nossos experimentos com Python para técnicos e engenheiros em Eletrônica.

A interface com o operador nos primeiros computadores era puramente física; através de chaves, botões, lâmpadas sinalizadoras e alarmes sonoros. Depois foi criado uma interface mais amigável com o operador com um *teletipo*, chamado de *TTY (Teletype)*; uma máquina de escrever eletromecânica usada para transmissão de dados à distância. Veja na figura abaixo como era essa interface do século passado.

Tinha teclado, 'monitor' (um rolo de papel) no centro e 'HD' (um leitor e perfurador de fita de papel, à esquerda). Na tampa à direita podia ser instalado um aparelho telefônico com disco para conexão remota de dados com outro TTY; o embrião da *internet* discada.

Nos anos 1980 este autor e o professor Paulo Brites daqui da *revista Antenna* trabalhamos juntos com esses equipamentos na estação de satélites da extinta *Embratel*.

Fonte: https://pt.wikipedia.org/wiki/Teletipo

Mas, hoje, interface gráfica com o usuário é nada mais que a tela de um programa num monitor *LCD*. Um painel colorido por onde o operador pode trocar informações com a máquina. Bons exemplos são as telas de controle das modernas máquinas mecânicas, como tornos mecânicos, máquinas CNC (Controle Numérico Computadorizado) e outros sistemas robóticos; também as telas do sistema operacional *Windows* da *Microsoft*; ou as telas dos aplicativos no seu *smartphone*, um computador de bolso equipado com rádio transceptor.

Aqueles botões, chaves, luzes piscantes e alarmes sonoros de antigamente ainda estão vivos; hoje virtuais, menores, multicoloridos e mais brilhantes; e podem ser facilmente remanejados de seus lugares ou substituídos por outros dispositivos como entradas de comando por voz, movimentos ou mesmo por ondas cerebrais (aliás, uma área interessantíssima!).

Conhecendo um novo editor de *scripts*: o VSCode

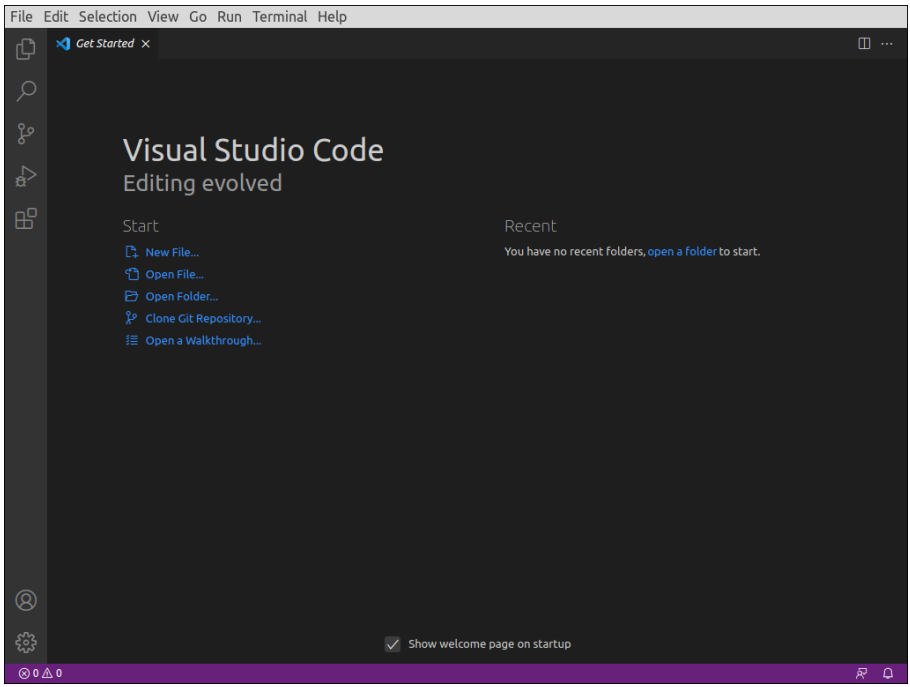
Até aqui em nossa série sobre Python, criamos e testamos nossos *scripts* na plataforma *on-line Google Colab*, um ambiente de desenvolvimento integrado próprio para essa linguagem que guarda nossos códigos remotamente numa nuvem. Mas existe um editor de códigos muito simples que pode ser instalado facilmente em qualquer sistema operacional, onde podemos testar nossos *scripts off-line* e guardá-los em nosso próprio computador.

Trata-se do *Visual Studio Code*, ou simplesmente *VSCode*, da *Microsoft*. Não se trata do *Visual Studio*, também da *Microsoft*, um *IDE* para desenvolvimento de aplicações nas linguagens *C, C++ e C#*; uma ferramenta mais pesada. O *VSCode* é uma ferramenta bem mais simples e com suporte para muitas outras linguagens, incluindo o Python.

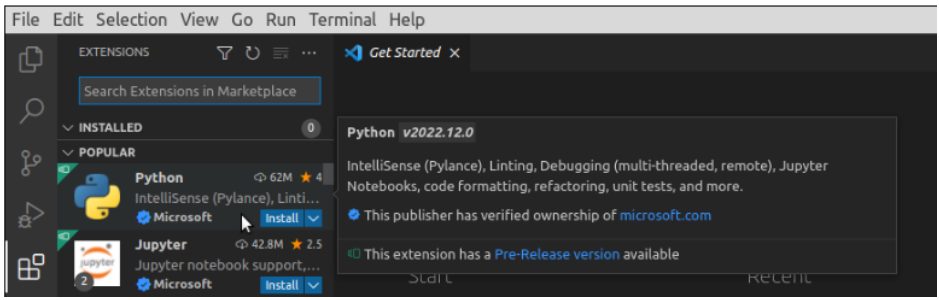
Uma das vantagens dessa plataforma é que, conforme formos evoluindo como programadores, podemos ir adicionando extensões para muitas outras funcionalidades. Por ser de código aberto, qualquer pessoa pode criar uma nova funcionalidade e publicá-la gratuitamente numa das páginas das muitas comunidades existentes na internet.

Aqui não vamos mostrar como se instala o *VSCode* na sua máquina; existe na internet dezenas de tutoriais que ensinam como fazer facilmente essa instalação em qualquer sistema operacional. Na lista de *sites we*b no final desse artigo colocamos algumas sugestões para o leitor.

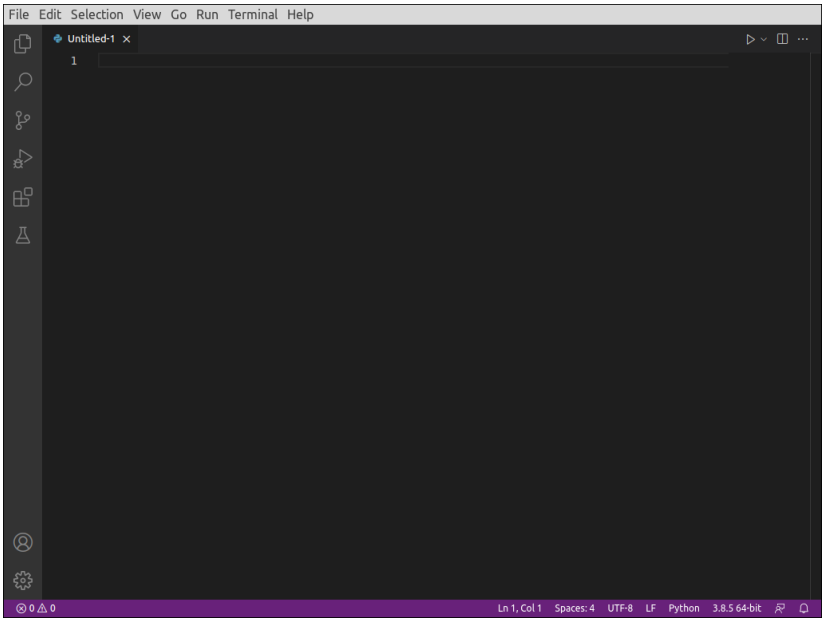
A tela limpa desse editor é minimalista, como podemos ver abaixo. É possível customizar o fundo da tela e também traduzir seus comandos para o português.



Para editarmos *scripts* em Python, nesse editor, devemos, antes de tudo, instalar a extensão para essa linguagem. Clique no ícone '*Extensions*', o quinto na coluna à esquerda, os quatro quadradinhos, e selecione a opção *Python*, que já deverá estar instalado no seu PC; como na tela a seguir.



Depois, feche a janela '*Get Started*' à direita (clicando no 'X' da janela) e o menu de extensões à esquerda clicando de novo no ícone '*Extensions*'. Essa é a tela prontinha para edição de nossos primeiros experimentos:



É uma tela bem simples, tem um menu enxuto na parte de cima com um nome sugerido para o primeiro projeto: *'untitled-1'*, uma coluna com 6+2 opções à esquerda e uma linha com algumas informações na parte de baixo. OK, agora vamos fazer um teste rápido para verificar se tudo está funcionando. Digite na área de edição, a partir da linha 1 mostrada, as seguintes 3 linhas de comandos:

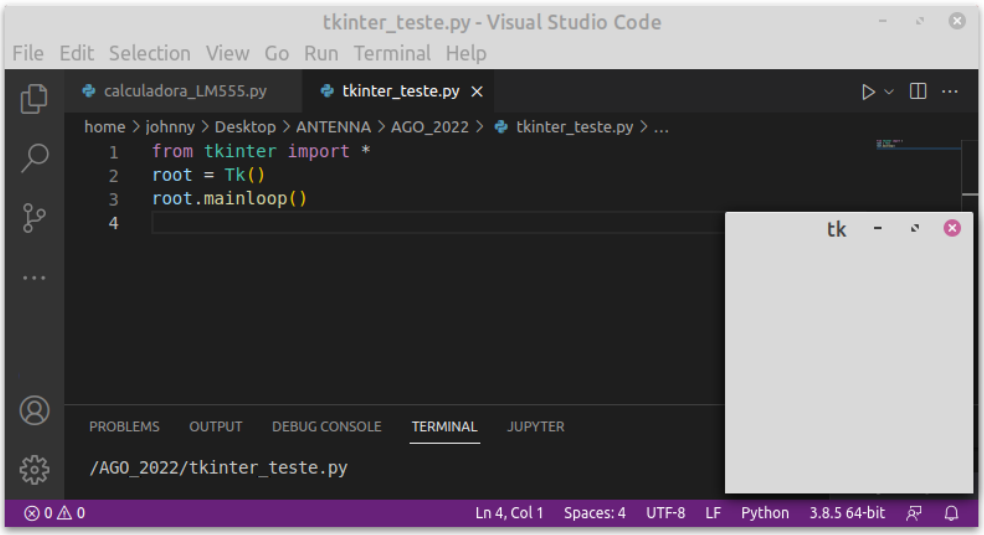
from tkinter import *
root = Tk()
root.mainloop()

Na primeira linha importamos toda a biblioteca *Tkinter* para nosso interpretador Python, criamos uma aplicação, *root*, e a mantemos aberta o tempo todo, com *root.mainloop()*.

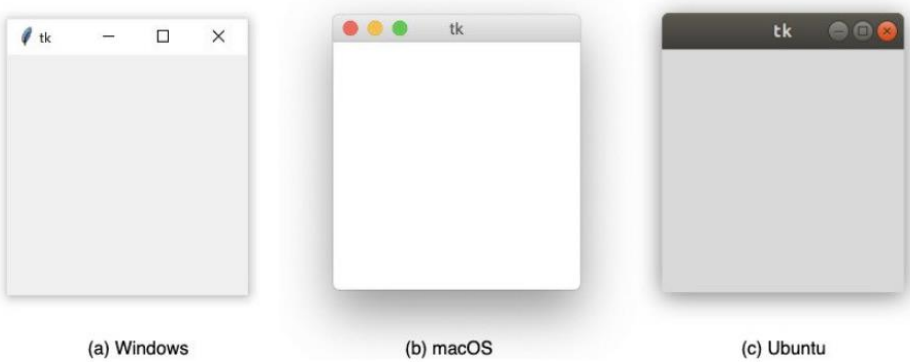
Depois, salve o *script* numa pasta na área de trabalho do seu computador digitando *'Cntl+S'*, ou clicando em *'File' > 'Save'*, e dando um nome com a extensão *.py* ao arquivo. Veja que aqui demos o nome de *'tkinter_teste.py'* ao nosso arquivo e o salvamos na subpasta *AGO_2022* da pasta *ANTENNA*.

Se tudo der certo, uma pequena janela gráfica deverá ser criada na tela do seu PC, com o título *'tk'* e os botões de controle dessa janela, como mostrado a seguir. Veja que um *Terminal* também surgiu na parte de baixo do editor, onde se pode ver o caminho do lugar onde o *script* está salvo.

É aqui que podem surgir as possíveis mensagens de erros e exceções em nosso programa.



Executamos esse *script* numa distribuição *Linux Mint*; as janelas gráficas podem ser ligeiramente diferentes se executado em outros sistemas operacionais. Veja as possíveis diferenças nas telas abaixo:



Janela em branco para cada sistema operacional

Fonte: https://programadoresbrasil.com.br/2021/05/tkinter-python-com-interface-grafica/

Se tudo funcionou, feche a janela gráfica de teste, clicando no seu botão correspondente. Depois também feche o Terminal e a janela de edição no *VSCode*, clicando no 'X' de cada um deles. Por fim, se quiser, apague o arquivo teste da sua pasta na área de trabalho.

Criando uma interface gráfica com Python

Na linguagem Python existe ao menos meia dúzia de bibliotecas para criação de interfaces com o usuário de um programa. Mas, vamos por ora nos ater somente à sua biblioteca padrão *Tkinter*. Essa biblioteca é nativa, vem junto com a instalação do interpretador Python no seu computador; portanto não é necessária qualquer instalação prévia para começar a usá-la; temos somente que importá-la logo no início de nosso *script*, como fizemos acima.

Antes de tudo cabe aqui uma breve descrição de alguns conceitos básicos sobre interfaces gráficas com o usuário, as chamadas *GUIs*. E lembrando sempre que: tudo no Python são objetos!

Container – uma alusão aos containers de navios, é um local delimitado onde podemos dispor objetos, os widgets;

Widgets – são componentes gráficos dentro do nosso container; são botões, chaves, caixas de textos, ícones e outros objetos;

Event Handlers – tratadores de eventos; são funções e métodos que são acionados somente quando por exemplo clicamos num botão;

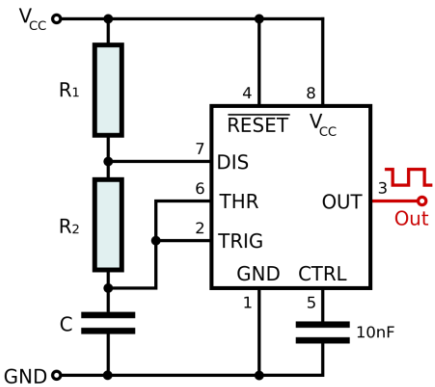
Event Loops – Laços de eventos; são scripts que ficam sendo executados o tempo todo à espera de um determinado evento; por exemplo, um contador de segundos.

Cremos que podemos comparar um *container* a uma placa de circuito impresso montada, onde os resistores, capacitores, transistores e LEDs são os *widgets*; os *tratadores de eventos* poderiam ser similarmente os botões e chaves mecânicas na placa; e os *laços de repetição* os diferentes circuitos ativos montados na placa; como amplificadores operacionais e portas lógicas digitais.

A biblioteca Tkinter (*Tk* + *interface*) na linguagem Python foi inspirada na biblioteca *Tk* de uma linguagem de programação de computadores, criada por um professor da Universidade da Califórnia nos EUA em 1988, a *Tcl* (*Tool Command Language*). Essa biblioteca se mostrou tão fácil de aprender e utilizar que logo foi implementada em muitas outras linguagens, como *Python*, *Perl, Ruby* e outras.

Vamos começar rascunhando numa folha de papel o *layout* da interface gráfica que temos em mente. Digamos que queremos criar o painel de uma simples calculadora da frequência de um oscilador astável com o cinquentenário circuito integrado LM-555. Essa calculadora vai nos dizer qual a frequência de oscilação dados os valores dos componentes do circuito.

O circuito básico de um oscilador astável com o temporizador LM-555 todo mundo já conhece, é o da figura abaixo.



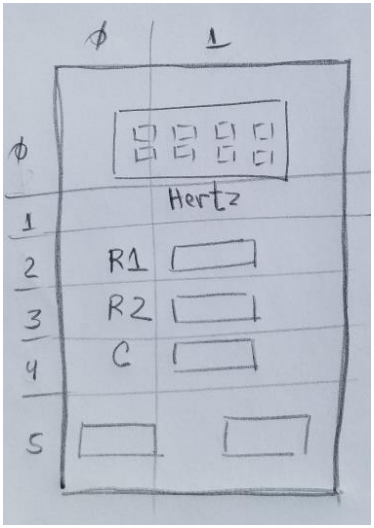
Fonte: https://commons.wikimedia.org/wiki/File:555_Astable_Diagram.svg

A frequência de saída desse oscilador é dada pelos valores de três componentes numa única malha: R1, R2 e C. Quando o circuito é alimentado, o pino 3 de saída está em Vcc e o capacitor C vai se carregando pela corrente que circula pelos resistores R1 e R2. Alcançado o valor de 2/3 de Vcc, um transistor interno do LM-555 conduz, e pelo pino 7 começa a descarregar o capacitor C através de R2 somente; nesse momento a saída do circuito é chaveada para um valor próximo de zero volt. Um outro transistor no pino 6 monitora a carga de C, e quando sua tensão cai para 1/3 de Vcc a saída no pino 3 volta para Vcc e todo o ciclo se repete. O que teremos da saída será uma onda quadrada pulsando entre Vcc e 0 volt. O pino 5 é conectado ao ponto de 2/3 do divisor de tensão interno, composto por 3 resistores de 5kΩ (daí o nome LM-555); normalmente deve ser desacoplado por um capacitor de 10nF.

A fórmula para determinar a frequência da onda quadrada no pino 3 desse circuito é:
*F = 1,44 / [C x (R1 + 2*R2)]*

Criando uma calculadora para o LM-555 com Python

Quais serão os *widgets* que devemos posicionar num *container* para essa nossa calculadora? Vamos lá: 1) um visor numérico onde podemos ler a frequência que foi calculada; 2) três campos para entrarmos com os valores de R1, R2 e C; 3) dois botões: um de 'OK' para calcular a frequência e outro para limpar os valores já inseridos; e, 4) as etiquetas (labels) de identificação desses widgets. Nosso layout poderia ser algo como mostrado na figura abaixo.



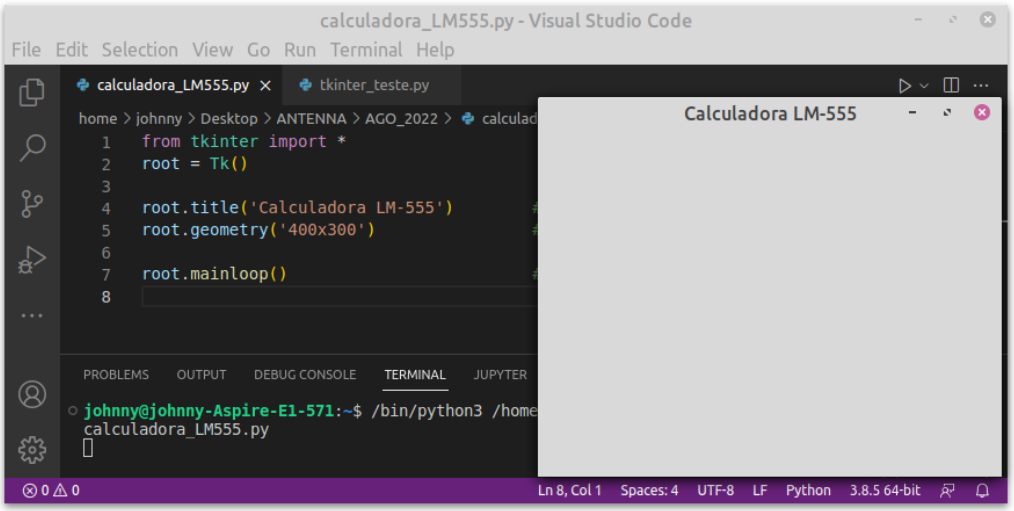
Para projetar nossa interface gráfica no *Tkinter* temos que dividir nosso *container* em 6 linhas horizontais e 2 verticais em nosso rascunho. Assim temos nossos *widgets* dispostos da seguinte forma dentro do *container*:

Linha	***Coluna***	***Widget***
0	***0***	***Visor numérico***
2	***0***	***texto: 'R1'***
3	***0***	***texto: 'R2'***
4	***0***	***texto: 'C'***
5	***0***	***Botão 'Limpar'***
1	***1***	***texto: 'Hertz'***
2	***1***	***Caixa de Entrada***
3	***1***	***Caixa de Entrada***
4	***1***	***Caixa de Entrada***
5	***1***	***Botão 'OK'***

Até aqui tudo bem? Agora no editor *VSCode* vamos começar nosso projeto. Vamos por partes, devagar. Primeiramente, copie as seguintes 5 linhas de código no editor e o salve numa pasta criada na área de trabalho do seu PC com o nome, por exemplo, *'calculadora_LM555.py'*. *Importante: Não esqueça que o arquivo salvo deve ter a extensão '.py'!*

Depois, execute esse *script* clicando no pequeno triângulo no topo do editor à direita (*'Run Python File'*). Veja o resultado logo abaixo.

```
from tkinter import *
root = tk.Tk()
root.title('Calculadora LM-555')
root.geometry('400x300')
root.mainloop()
```

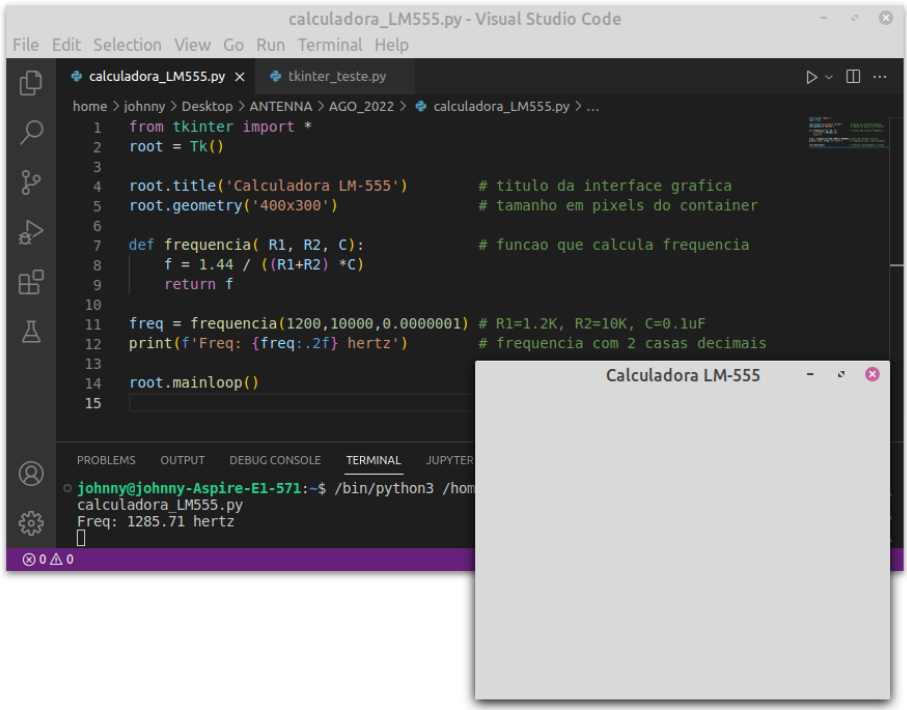


Criamos uma janela (container) com o título *'Calculadora LM-555'* e o dimensionamos para 400 pixels de largura e 300 pixels de altura.

Mais tarde, se necessário, podemos redimensionar nosso container para acomodar todos os *widgets*. Se você esquecer de digitar a última linha, *root.mainloop()*, o programa vai ser executado uma só vez e a janela nem vai aparecer na tela do seu PC; é essa linha que mantem a janela visível.

Antes de posicionar no nosso container os *widgets*, temos que testar uma função em Python que calcula a frequência de saída do nosso oscilador com o LM-555. Acrescente as seguintes linhas no editor, antes da linha que executa continuamente o *script* (*'root.mainloop()'*) e veja o resultado no Terminal, com os valores R1=1.2kΩ, R2=10kΩ e C=0.1uF, na tela abaixo:

```
def frequencia( R1, R2, C):
    f = 1.44 / ((R1+R2) *C)
    return f
freq = frequencia(1200, 10000, 0.0000001)
print(f'Freq: {freq:.2f}' hertz)
```



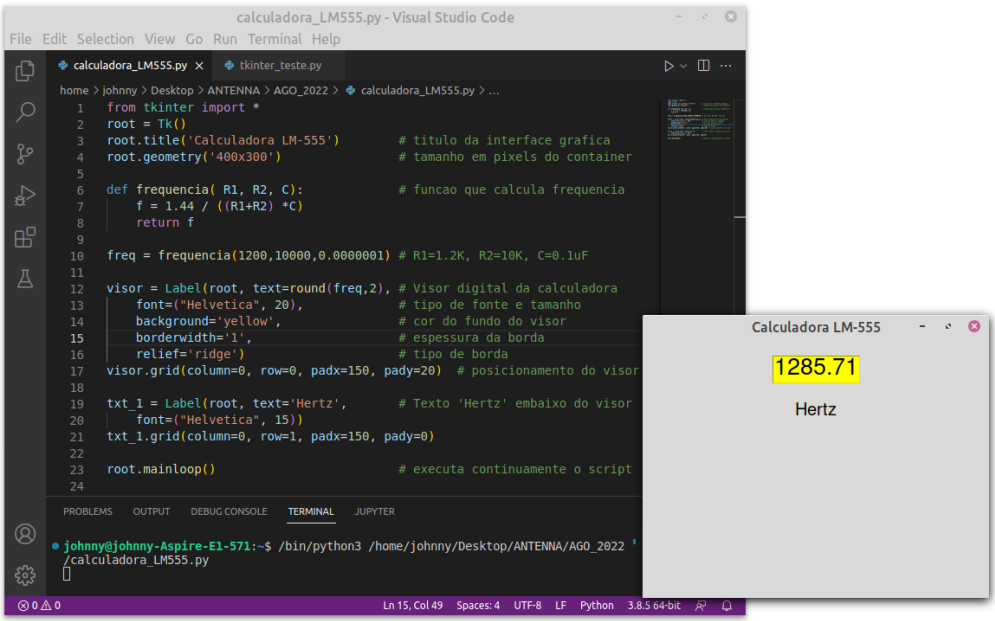
O que temos nesse *script*? Na primeira linha importamos tudo da biblioteca *Tkinter*: todos os seus métodos e funções. Na segunda linha criamos um container vazio, chamado de *'root'*, a aplicação raiz com barra de título e seus botões; como fizemos no primeiro *script* de teste. Depois demos um título à aplicação e definimos seu tamanho em *pixels*.

Logo depois, criamos uma função para calcular a frequência do nosso circuito astável com o LM-555 e a testamos quando chamamos essa função e passamos os parâmetros necessários (R1, R2 e C). Depois mandamos imprimir no Terminal o retorno dessa função. Com os valores de teste, R1=1.2kΩ, R2=10kΩ e C=0.1uF, nosso *script* calculou uma frequência de oscilação para o circuito de 1.285,71 Hz. Confira na saída do *Terminal* na parte de baixo do editor *VSCode*. Por enquanto nosso container está vazio, tem somente o título de '*Calculadora LM-555*' e seus botões padronizados no topo.

Agora sim, podemos começar a posicionar cada objeto do nosso rascunho dentro da janela criada para a calculadora. O primeiro deles será o visor numérico que vai mostrar a frequência calculada conforme os valores dos componentes do circuito, e uma etiqueta com a palavra 'Hertz'. Usaremos dois *widgets Label()* do *Tkinter*, são duas caixas de texto.

Copie e cole as seguintes linhas de código no editor *VSCode* e salve-o com *'Cntl+S'*. Depois execute o *script* clicando no pequeno triângulo na parte superior da tela. Veja o resultado na figura abaixo.

```
from tkinter import *
root = Tk()
root.title('Calculadora LM-555')
root.geometry('400x300')

def frequencia( R1, R2, C):
    f = 1.44 / ((R1+R2) *C)
    return f

freq = frequencia(1200,10000,0.0000001)

visor = Label(root, text=round(freq,2), font=("Helvetica", 20),
        borderwidth='1', relief='ridge')
visor.grid(column=0, row=0, padx=150, pady=20)

txt_1 = Label(root, text='Hertz', font=("Helvetica", 15))
txt_1.grid(column=0, row=1, padx=150, pady=0)

root.mainloop()
```



O *widget Label()* é uma *classe* do *Tkinter* usada para mostrar textos e imagens na nossa aplicação gráfica. Sua sintaxe é bem simples:

Label(container, opções)

O primeiro parâmetro é o nome da janela, que criamos logo no início do *script*, onde será inserida a caixa de texto. São muitas as opções, até mesmo o texto é opcional, podemos criar uma caixa vazia com *text=' '*. Aqui definimos o tipo de fonte e seu tamanho, o fundo da caixa e o tipo de espessura. Também temos que definir onde essa caixa de texto será posicionada na janela com o método *.grid()* do *Tkinter*, que utiliza o conceito de linhas e colunas; por isso tivemos que criar um primeiro rascunho.

Por fim, também criamos uma caixa de texto com a palavra *'Hertz'* e a posicionamos logo abaixo da primeira, na segunda linha.

Bem, caros leitores, esse nosso nono artigo está ficando bem mais longo do que pensávamos, e ainda temos que montar e testar todos os outros objetos (*widgets*) na janela da nossa calculadora; assim, propomos dar uma parada por ora no nosso projeto e continuarmos no próximo trabalho.

Sugerimos que o leitor aproveite e se familiarize com o excelente editor *VSCode* e também busque conhecer pela internet os outros *widgets* do *Tkinter*. Abaixo sugerimos alguns links.

Até lá!

Instalação do VSCode no Windows, MacOS e Linux:
https://www.treinaweb.com.br/blog/instalacao-do-vs-code-no-windows-linux-e-macos/

https://www.youtube.com/watch?v=ctcDfKYrzOQ

Extensões para o VSCode:
https://www.treinaweb.com.br/blog/vs-code-melhores-extensoes-para-front-end-parte-1/

Um curso de Tkinter:
https://python-course.eu/tkinter/

João Alexandre Silveira*

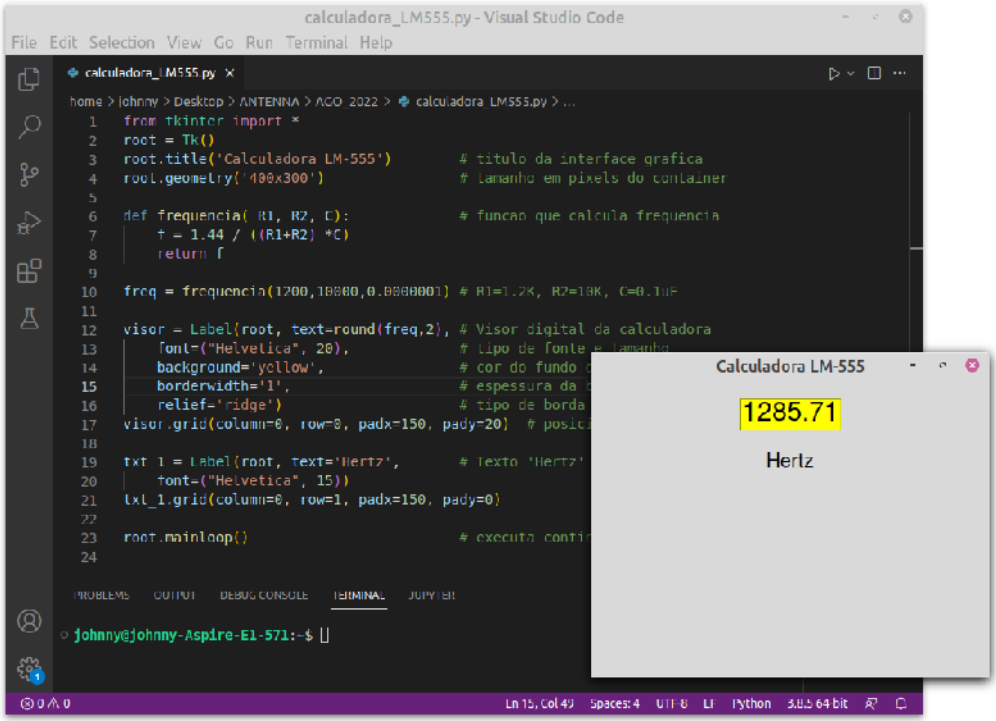
Hello World! Ou melhor: Olá, caros leitores da revista Antenna! Como este artigo chegamos à décima parte de nossa série de experimentos com a linguagem Python para técnicos e engenheiros em Eletrônica.

No trabalho anterior, publicado na revista Antenna de agosto passado, tivemos que interromper subitamente nossa explanação sobre interfaces gráficas com a biblioteca *Tkinter* porque o texto estava ficando extenso demais; assim, optamos por dividi-lo em duas partes: esta é a segunda parte da nona parte, ou a décima parte de toda a série. Explicamos ou confundimos?

Naquela nona parte conhecemos um editor minimalista de *scripts*, o *VSCode* da *Microsoft*. Para aqueles leitores que não se lembram, aqui vai de novo a cara dele, do editor, com o último *script* que criamos e o produto de sua execução, uma tela *GUI* (*Graphical User Interface*,) ainda insípida, de uma calculadora para o circuito de um oscilador astável com o temporizador LM-555.

Em 23 linhas de código em Python, importamos toda a biblioteca *tkinter*, criamos uma janela de 400x300 *pixels* e uma função, que recebe como parâmetros os valores dos dois resistores e do capacitor que determinam a frequência de saída do nosso circuito; também criamos dois *widgets*; são duas caixas de mensagens ou etiquetas (*Labels*), batizadas de 'visor' e 'txt_1'.

*Autor do livro "Experimentos com o Arduino", disponível em www.amazon.com.br



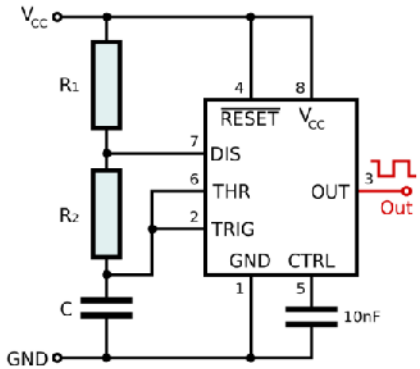
Essa primeira interface gráfica mostra somente o resultado do retorno de uma função em Python que criamos na linha 6, dados os valores de R1, R2 e C. Conforme visto naquela nona parte no mês passado, discorremos sobre o que é uma interface: a 'superfície' que forma o limite comum entre dois corpos, espaços e tempos. Será também o presente uma interface entre o passado e o futuro? Eis aí uma questão metafísica.

Sugerimos agora ao caro leitor rever a primeira parte desse artigo, onde falamos como importar a biblioteca *Tkinter* e como é a aparência da janela que será criada conforme o sistema operacional instalado no seu PC. Todas as janelas *GUI* aqui apresentadas são aquelas criadas numa distribuição *Linux Mint*. Num ambiente *Windows* ou *MacOS* as janelas serão ligeiramente diferentes, porém com a mesma funcionalidade. Dito isso, vamos adiante.

O Container Calculadora com o LM-555

Já sabemos que *container,* numa interface gráfica, é uma área delimitada onde dispomos os nossos *widgets,* que são os objetos que queremos usar, como botões, etiquetas e caixas de entrada de textos.

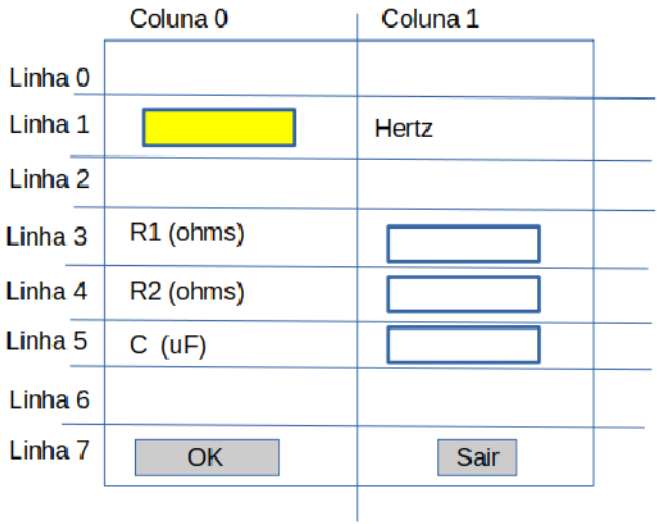
Aqui o nosso container é a janela gráfica de uma calculadora, que nos mostra a frequência de saída de um circuito astável padrão, montado com o circuito integrado LM-555. Veja novamente o circuito:



Fonte: https://commons.wikimedia.org/wiki/File:555_Astable_Diagram.svg

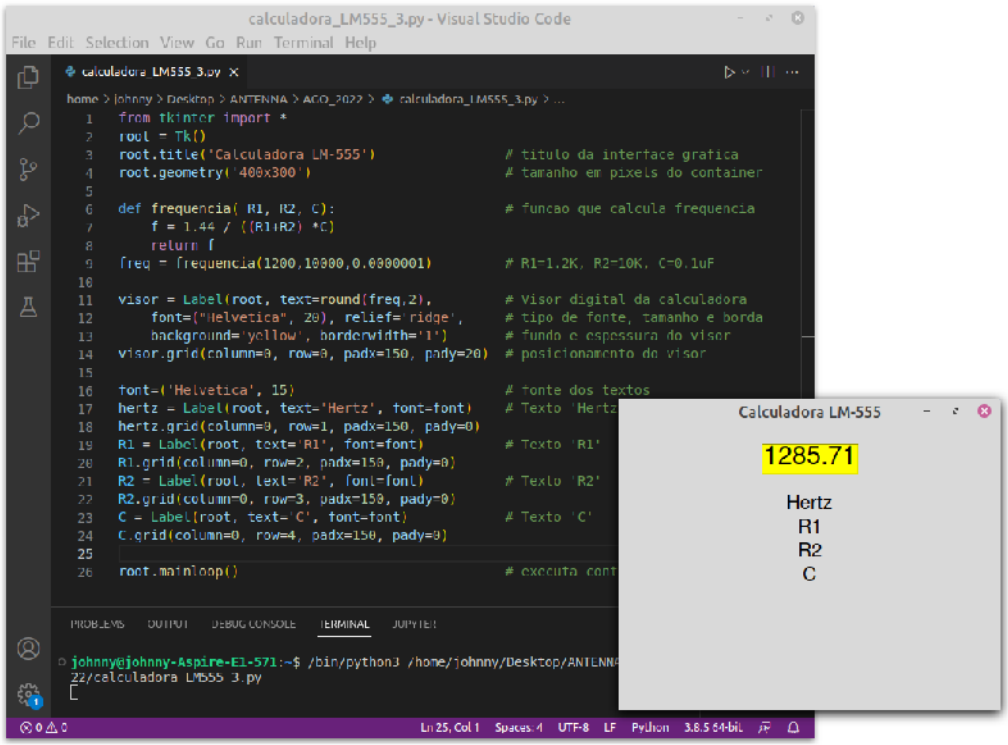
Como já tínhamos visto, nossa calculadora LM-555 terá como *widgets* um mostrador numérico; três campos para entrada dos valores de R1, R2 e C; um botão de '*OK*' para confirmar as entradas desses valores; e, mudamos, um botão '*Sair*' para fechar o programa, além de etiquetas (*labels*) de identificação desses *widgets*.

Refizemos o *layout* de nossa calculadora, que agora será como mostrado a seguir:



Em nosso rascunho, todos os *widgets* são dispostos numa matriz de 6 linhas por 2 colunas. Na primeira coluna temos o mostrador, as etiquetas R1, R2 e C e o botão 'OK'. Na segunda coluna, o texto 'Hertz', as caixas de entradas para R1, R2 e C e o botão 'Sair'.

Vamos agora abrir nosso *script*, que salvamos como '*Calculadora_LM-555_3.py*', a versão 3, no editor *VSCode* e criar as outras etiquetas (*Labels*), conforme o *layout* que rabiscamos acima.



Note que, no *script* acima, temos somente a coluna 0 em nosso container; com somente o visor numérico, que mostra a frequência calculada do oscilador, e todas as etiquetas identificadoras dos *widgets*. Vamos agora criar a coluna 1 e nela colocar 3 *widgets* do tipo caixas de entrada de texto, para receber os valores de R1, R2 e C.

O projeto final da Calculadora com o LM-555

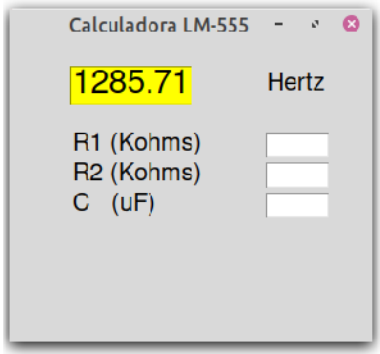
Até agora, vimos que o *widget Label()* é usado para exibir textos prontos como forma de identificação ou etiquetas de outros *widgets* dentro do container.

Para inserir valores pelo usuário manualmente, para processamento posterior, usamos o *widget Entry()*. Vejamos isso no nosso programa *calculadora _LM555_4.py* versão 4 abaixo.

```
from tkinter import *                  # importa toda a biblioteca tkinter
root = Tk()                            # cria um container  grafico 'root'
root.title('Calculadora LM-555')       # título da interface gráfica
root.geometry('300x230')               # tamanho em pixels do container

def frequencia( R1, R2, C):            # funcao que calcula frequencia
    f = 1.44 / ((R1+R2) *C)
    return f
freq = frequencia(1200,10000,0.0000001)     # R1=1.2K, R2=10K, C=0.1uF

Label(root, text='').grid(column=0, row=0)      # linha 0 vazia
visor = Label(
    root, text=round(freq,2),          # Visor digital da calculadora
    justify=LEFT, anchor='w',          # posiciona visor na esquerda
    font=("Helvetica", 20), relief='ridge',     # tipo de fonte, tamanho e borda
    background='yellow', borderwidth='1'        # fundo e espessura do visor
)
visor.grid(sticky = W, column=0, row=1, padx=50)     # posicionamento do visor

font=('Helvetica', 15)                 # fonte dos textos

hertz = Label(
    root, text='Hertz',
    font=font, justify=LEFT, anchor='w'
)
hertz.grid(sticky = W, column=1, row=1)

Label(root, text='').grid(column=0, row=2)      # linha 2 vazia
R1 = Label(
    root, text='R1 (Kohms)', font=font,        # etiqueta para R1
    justify=LEFT, anchor='w'
)
R1.grid(sticky = W, column=0, row=3, padx=50)

R2 = Label(
    root, text='R2 (Kohms)', font=font,        # etiqueta para R1
    justify=LEFT, anchor='w'
)
R2.grid(sticky = W, column=0, row=4, padx=50)

C = Label(
    root, text='C   (uF)', font=font,          # etiqueta para C
    justify=LEFT, anchor='w'
)
C.grid(sticky = W, column=0, row=5, padx=50)

eR1 = Entry(root, width=6)             # caixa de entrada para R1
eR1.grid(sticky = W, column=1, row=3)

eR2 = Entry(root, width=6)             # caixa de entrada para R2
eR2.grid(sticky = W, column=1, row=4)
```

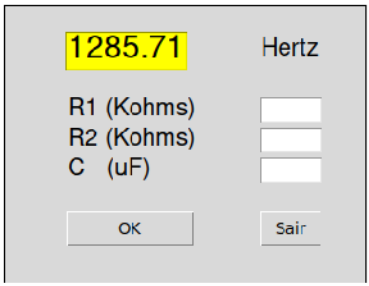
```
eC =  Entry(root, width=6)                    # caixa de entrada para C
eC.grid(sticky = W, column=1, row=5)

Label(root, text='').grid(column=0, row=6)          # linha 6 vazia

root.mainloop()                               # executa continuamente o script
```

E aqui está a tela gráfica criada com o *script* acima:



Quase pronto. Vamos criar agora os dois botões da nossa calculadora. Acrescente ao código acima as seguintes linhas, antes da última linha, e veja o resultado logo abaixo:

```
ok = Button(root, text='OK',  width=10, command=frequencia)      # cria botao 'OK'
ok.grid(sticky = W, column=0, row=7, padx=50)

sair = Button(root, text='Sair', command=root.quit)              # cria botao 'Sair'
sair.grid(sticky = W, column=1, row=7)

root.mainloop()                                  # executa continuamente o script
```



Por fim, vamos fazer os ajustes em nosso programa para que nossa calculadora possa mostrar no seu visor a frequência de saída do oscilador, conforme os valores de R1, R2 e C, entrados pelo usuário nas 3 caixas de entradas.

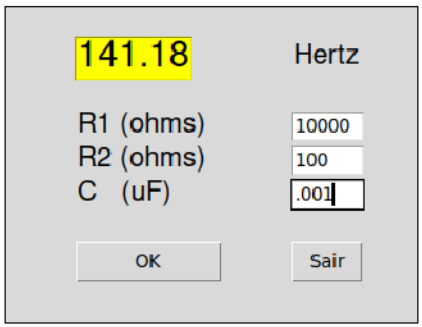
Como dissemos acima, o widget *Entry()* é usado para digitarmos dados manualmente numa caixa de textos. Para nosso *script* capturar esse dado entrado e processá-lo numa função temos que usar o método *get()*.

Para não nos alongarmos (de novo), aqui vai o programa completo onde, em cada linha, existe um comentário (logo depois do '#') explicando sua função:

```
from tkinter import *                    # importa toda a biblioteca tkinter
root = Tk()                              # cria um container  grafico 'root'
root.title('Calculadora LM-555')                 # titulo da interface grafica
root.geometry('300x230')                 # tamanho em pixels do container
root.resizable(False, False)             # desabilita redimensionamento da janela

font=('Helvetica', 15)                   # fonte dos textos

def visor(f):                            # funcao que cria visor default
   print(f)
   visor = Label(
      root, text=round(f,2),             # Visor digital da calculadora
      justify=LEFT, anchor='w',          # posiciona visor na esquerda
      font=("Helvetica", 20), relief='ridge',      # tipo de fonte, tamanho e borda
      background='yellow', borderwidth='1'  # fundo e espessura do visor
   )
   visor.grid(sticky = W, column=0, row=1, padx=50)  # posicionamento do visor

visor(.0000)                             # chama funcao visor default

def frequencia():                              # funcao que atualiza visor
      r1 = int(eR1.get())                # captura valor R1
      r2 = int(eR2.get())                # captura valor R2
      c =  float(eC.get())               # captura valor C
      f = (1.44 / ((r1+2*r2) *c))         # formula da frequencia
      #print(f)
      visor(f*1000)                      # mostra valor em hertz no visor

hertz = Label(                           # label 'Hertz'
      root, text='Hertz',
      font=font, justify=LEFT, anchor='w'
)
hertz.grid(sticky = W, column=1, row=1)

Label(root, text='').grid(column=0, row=2)          # linha 2 vazia
R1 = Label(
      root, text='R1 (Kohms)', font=font,  # label 'R1'
      justify=LEFT, anchor='w'
)
R1.grid(sticky = W, column=0, row=3, padx=50)
R2 = Label(
      root, text='R2 (Kohms)', font=font,  # label 'R2'
      justify=LEFT, anchor='w'
)
R2.grid(sticky = W, column=0, row=4, padx=50)

C = Label(
      root, text='C   (uF)', font=font,    # label 'C'
      justify=LEFT, anchor='w'
)
C.grid(sticky = W, column=0, row=5, padx=50)
```

```
eR1 = Entry(root, width=6)                  # caixa de entrada para R1
eR1.grid(sticky = W, column=1, row=3)
eR1.focus()                                 # foco inicial na caixa eR1

eR2 = Entry(root, width=6)                  # caixa de entrada para R2
eR2.grid(sticky = W, column=1, row=4)

eC =  Entry(root, width=6)                  # caixa de entrada para C
eC.grid(sticky = W, column=1, row=5)

Label(root, text='').grid(column=0, row=6)      # linha 6 vazia

Label(root, text='').grid(column=0, row=0)          # linha 0 vazia

ok = Button(root, text='OK',  width=10, command=frequencia) # cria botao 'OK'
ok.grid(sticky = W, column=0, row=7, padx=50)

sair = Button(root, text='Sair', command=root.quit)  # cria botao 'Sair'
sair.grid(sticky = W, column=1, row=7)

root.mainloop()                             # executa continuamente o script
```

Lá no topo do programa, na linha 5, desabilitamos o redimensionamento da janela com *root.resizable(False, False)*. Depois criamos uma função '*visor*', que recebe como parâmetro o valor da frequência calculada em uma outra função, a função '*frequencia*', com os valores de R1, R2 e C. Essa primeira função atualiza o valor mostrado no visor da calculadora depois que clicamos no botão 'OK'. Após cada entrada dos valores de R1, R2 e C, pressione a tecla '*Tab*' no seu PC; depois clique no botão '*OK*' para ver a frequência no visor amarelo. O botão '*Sair*' termina o programa e fecha a janela da calculadora.

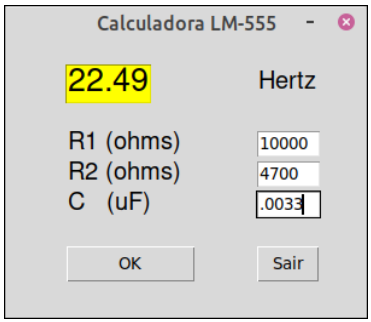


A tela final do nosso projeto está mostrada acima. Com os valores indicados, R1=10KΩ, R2=1KΩ e C=.001μF, a frequência do nosso oscilador astável será de 141,18 Hz.

Ufa! Esse foi o mais longo experimento com Python mostrado nessa nossa série. Num próximo trabalho vamos ver como transformamos esse *script* num programa executável. Até lá!

João Alexandre Silveira*

Na parte 10 desta nossa série sobre a linguagem Python, criamos uma *interface GUI* com a biblioteca *tkinter*, a *calculadora LM-555*. A cara dela aparece aqui embaixo. Básica, com um painel só com o essencial: um visor para mostrar a frequência de saída de um oscilador astável montado com o circuito integrado temporizador LM-555; e três entradas para os valores da malha de oscilação do circuito. Dois botões: um confirma os valores digitados e o outro encerra o programa.



Com o que aprendemos até aqui, podemos incorporar outras funções a nossa calculadora; como um outro visor que mostre o ciclo de trabalho (*duty cycle*) da onda quadrada gerada na saída do oscilador. Podemos também remanejar algumas linhas de código no *script* em Python para que nossa calculadora aceite como uma das entradas o parâmetro *frequência* e calcular o valor do capacitor, por exemplo.

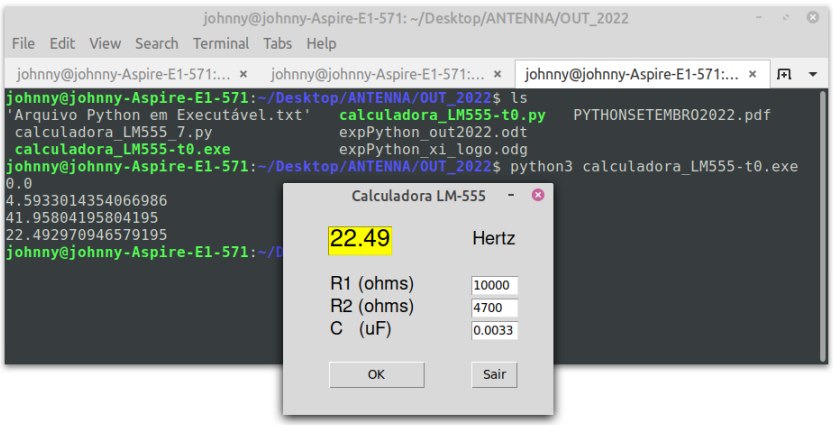
***Autor do livro "Experimentos com o Arduino", disponível em** www.amazon.com.br

Mas, veja, toda vez que quisermos usar essa calculadora, deveremos abrir primeiro um interpretador de Python, no nosso caso o editor *VSCode*, carregar nele o *script* e então executar o aplicativo. O que queremos é uma calculadora como aquela do sistema operacional instalado em nosso PC, que é só dar dois cliques num ícone e, *voilà!*, uma calculadora aparece no centro do monitor.

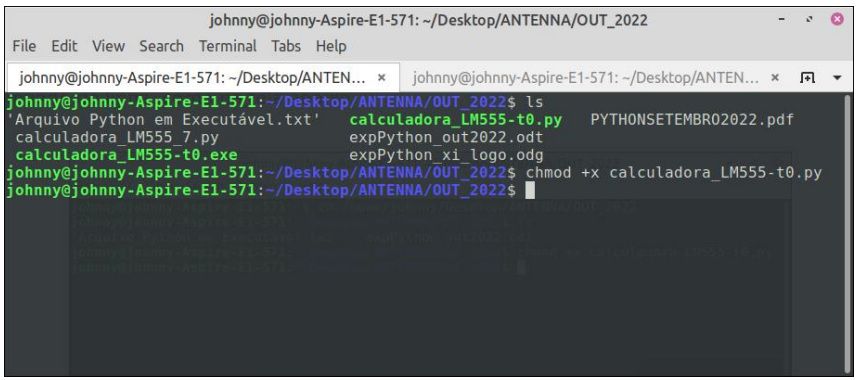
É disso que trataremos nesta 11ª parte de nossa jornada através do mundo da linguagem Python; que será mais breve, dessa vez sem códigos para testar no *VSCode*.

Como converter '.py' em '.exe'

Num ambiente Linux, facilmente podemos criar um *arquivo* executável a partir de um *script* Python em dois passos: copie seu *script* como o mesmo nome, porem troque a extensão '*.py*' para '*.exe*'; e no terminal, digite '*python3*' antes do nome do arquivo com a nova extensão.



Ou de modo pitônico, num terminal torne seu arquivo '*.py*' executável através de um simples comando de linha com *chmod*, como na tela abaixo.



Mas existe uma forma, através de uma *GUI,* de criar um arquivo executável em qualquer sistema operacional, a partir de um *script* em Python: utilizando-se o programa conversor '*auto-py-to-exe*'. O conversor '*auto-py-to-exe*' é uma interface, uma janela aberta num navegador *web*, como o *Chrome* da *Google*, que tem por trás a biblioteca Python *pyinstaller*.

De forma bem sucinta: essa biblioteca *pyinstaller* compila um aplicativo escrito em Python e todas as suas dependências em um único pacote.

Esse pacote é um arquivo executável, como qualquer outro aplicativo. O usuário final do programa não precisará ter um interpretador ou qualquer outro módulo instalado em seu PC.

Para fazer a conversão, o *pyinstaller* lê o arquivo texto em Python escrito pelo programador, identifica todas as necessidades para que esse *script* se torne um arquivo executável, como outras bibliotecas e módulos, e, junto com um interpretador, compila tudo num só arquivo. E opcionalmente também seus vários módulos separados em pastas.

Uma vez instalada no PC a biblioteca *pyinstaller*, compilamos de forma bem simples um *script* em Python pelo terminal em uma só linha de comando:

pyinstaller nome_do_script.py

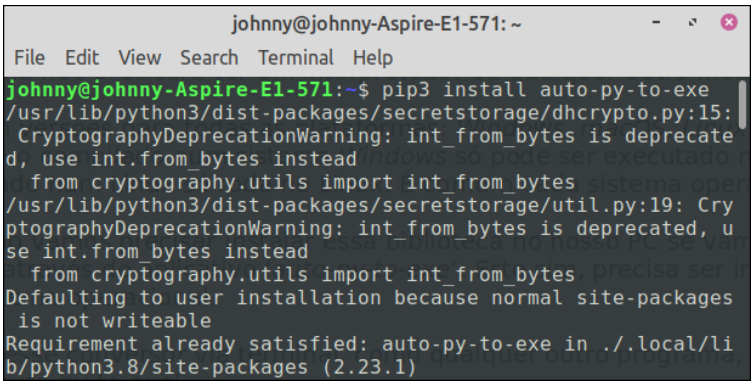
Mas existem dezenas de opções e configurações para a geração do arquivo final com esse conversor multiplataforma; como inclusão de *ícone* e arquivos de mídia e do tipo *.xlsx* ou *.csv*, que podem ser testadas pelo leitor nos *links* sugeridos no final do artigo.

Existem versões do *pyinstaller* para todas as plataformas: *Windows, MacOS, Linux* e outras. Porém, um aplicativo compilado num sistema *Windows* só pode ser executado nesse sistema; um outro compilado num Linux só roda no Linux.

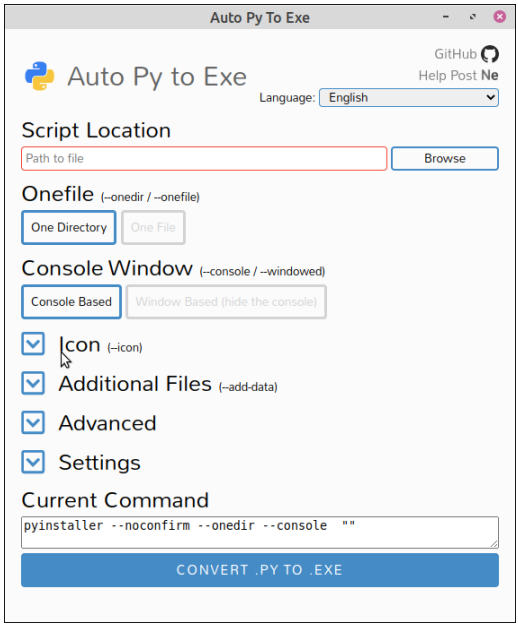
É comum cada sistema operacional ter seus programas nativos.

Mas não vamos precisar instalar essa biblioteca *pyinstaller* no nosso PC se vamos converter nossos *scripts* através do aplicativo gráfico '*auto-py-to-exe*'. Este, sim, precisa ser instalado via o terminal do seu sistema operacional.

Instale essa interface *GUI* via terminal, como qualquer outro programa, com o comando '*pip install auto-py-to-exe*'.

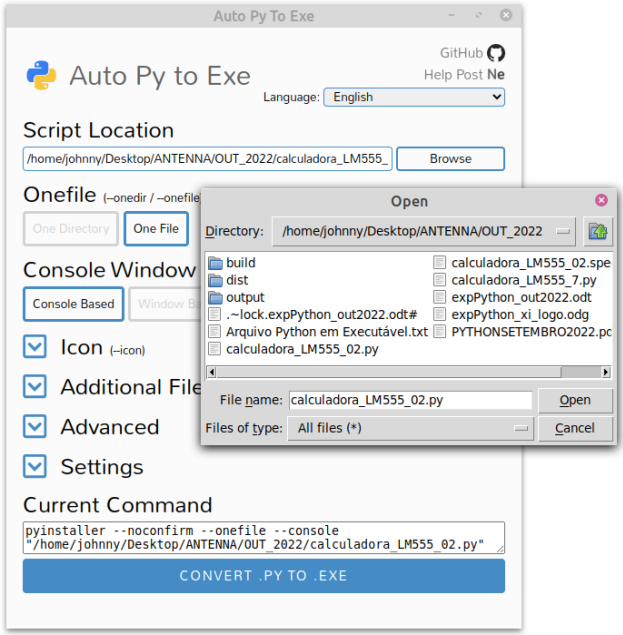


Concluída a instalação, ainda no terminal, abra o programa digitando ***auto-py-to-exe***. Após alguns segundos, a seguinte janela deverá se abrir como uma página *web* no seu navegador de internet.

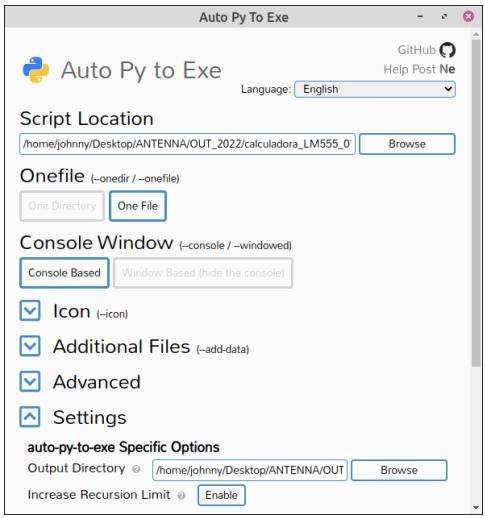


Você pode mudar a língua *default* para *Português Brasileiro* no botão à direita no topo da tela.

Logo embaixo, em '*Script Location*', clique no botão '*Browse*', à direita. Na pequena tela que surgir, mude a opção em '*Files of type:*' para '*All files (*)* e procure pelo seu código Python, que deverá estar numa pasta no seu PC. Encontrado seu arquivo '*.py*', clique uma vez nele e no botão '*Open*' dessa pequena tela.



De volta à tela principal '*Auto Py To Exe*', clique no botão '*One File*' para criar um arquivo executável único. Em '*Settings*', figura abaixo, informe onde deverá ser gravado o novo arquivo executável.

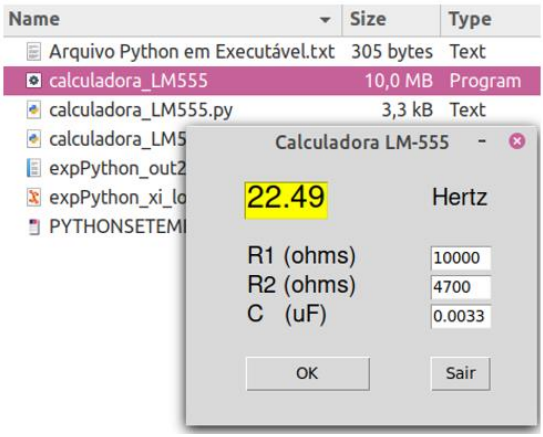


Por fim, clique lá embaixo no grande botão azul '*Convert .py to .exe*'. Aguarde alguns segundos e após a conversão, no final da tela, clique no outro botão azul '*Open Output Folder*' e veja onde está o arquivo compilado; como mostrado na tela a seguir:

Name	Size	Type
Arquivo Python em Executável.txt	305 bytes	Text
calculadora_LM555_01	10,0 MB	Program
calculadora_LM555_01.py	3,3 kB	Text
calculadora_LM555_02.py	3,3 kB	Text
calculadora_LM555_02.spec	841 bytes	Text
calculadora_LM555_7.py	3,3 kB	Text
expPython_out2022.odt	1,2 MB	Document
expPython_xi_logo.odg	118,5 kB	Image
PYTHONSETEMBRO2022.pdf	1,3 MB	Document

Aqui, onde foi configurado, está o aplicativo executável em um único arquivo, chamado de '***calculadora_LM55_01***', sem qualquer extensão. O segundo da lista ao lado, com um ícone próprio.

Vamos testá-lo? Dê um duplo clique sobre esse arquivo. Funcionou! A janela do nosso aplicativo está ativa na tela do monitor, esperando as entradas dos valores de R e C para calcular a frequência do LM-555.



Agora, você pode distribuir seus aplicativos executáveis. Crie um atalho para esse programa e o coloque na área de trabalho do seu PC. Pronto. Como qualquer outro programa, nossa calculadora LM-555 pode ser executada com somente dois cliques.

Para fechar nosso artigo deste mês, desafiamos nossos leitores a criar outras calculadoras semelhantes; por exemplo, uma para a aplicação da lei de Ohm; ou uma para cálculos dos valores de um circuito padrão montado com o também clássico opamp LM-741. Até breve!

LINKS:

Auto-py-to-exe: ***https://pypi.org/project/auto-py-to-exe/***

https://proxlight.medium.com/how-to-convert-py-to-exe-step-by-step-guide-82e9e9a8984a

pyinstaller: **https://datatofish.com/executable-pyinstaller/**

João Alexandre Silveira*

Chegamos, com esta décima segunda parte de nossos *Experimentos com a Linguagem Python para Técnicos em Eletrônica,* a um ano inteiro de publicação aqui na revista *Antenna*. Com esta parte XII chegamos ao fechamento de um ciclo, como profissionais de *hardware,* descobrimos que *software* é fácil, e que hoje os dois se entrelaçam como na figura do *yin-yang*. Pensamos que conseguimos cobrir nesta nossa série todos os comandos e funções básicas do Python.

Para fechar este um ano de experimentos com *software*, selecionamos 8 projetos simples e incrivelmente úteis para que nosso leitor possa conhecer alguns dos inúmeros campos de aplicação dessa versátil e fácil linguagem de programação de sistemas com processadores digitais. Vamos a eles.

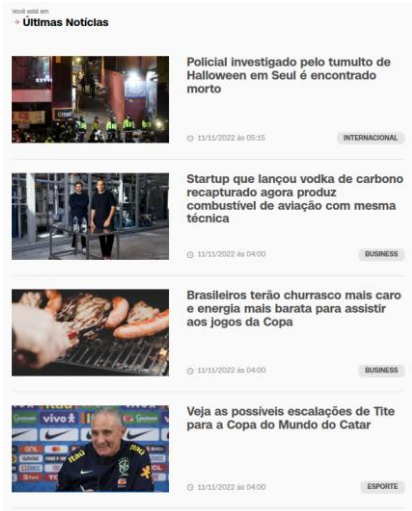
1. Scrapper de notícias de jornais

Que tal se manter atualizado a qualquer momento com as manchetes das últimas notícias, as chamadas '*top news headlines*', postadas nos principais canais de informações sobre política, economia, guerra na Ucrânia, esporte, tecnologia e saúde, na internet?

Criamos um *scrapper* para listar somente as manchetes de um jornal ou canal de notícias na web. Para executar esse *script* você deve instalar as bibliotecas *'requests'* e *'bs4',* com *pip install requests e pip install bs4* pelo terminal do seu sistema operacional.

*__Autor do livro "Experimentos com o Arduino", disponível em__ www.amazon.com.br

Já vimos que bibliotecas são coleções de métodos e funções prontinhas para uso direto no Python; são contribuições de desenvolvedores que disponibilizam graciosamente na *web* suas invenções. Aqui temos a tela capturada do canal da CNN em um dado dia e hora.



No *script* abaixo, *'headline_news.py'*, depois de importar as duas bibliotecas, para nossas experiências, criamos três *tuplas*, *'cnn'*, *'veja'* e *'yahoo'*, que guardam o endereço *web* desses canais de notícias e a *tag html,* que define o tamanho da fonte do título das manchetes nesses canais.

```
headline_news.py
home > johnny > Desktop > ANTENNA > NOV_2022 > headline_news.py > ...
1   # Scrapper de notícias de jornais
2   # pip install requests
3   # pip install bs4
4   import requests
5   from bs4 import BeautifulSoup
6   cnn = 'https://www.cnnbrasil.com.br/ultimas-noticias/', 'h3'
7   veja = 'https://veja.abril.com.br/ultimas-noticias/','h2'
8   yahoo = 'https://br.noticias.yahoo.com/','h3'
9
10  response = requests.get(cnn[0])
11  html = BeautifulSoup(response.text, 'html.parser')
12
13  for news in html.find_all(cnn[1] ):
14      print(news.text)
15  # END

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

Policial investigado pelo tumulto de Halloween em Seul é encontrado morto
Startup que lançou vodka de carbono recapturado agora produz combustível de aviação com mesma técnica
Brasileiros terão churrasco mais caro e energia mais barata para assistir aos jogos da Copa
Veja as possíveis escalações de Tite para a Copa do Mundo do Catar
Gal Costa será velada na Assembleia de SP a partir das 9h desta sexta-feira (11)
Com Lula em SP, conselho político de transição se reúne no CCBB nesta sexta
COP27 se prepara para entrar na reta final cercada de expectativas e desafios
Enem 2022 começa neste domingo (13); veja detalhes sobre o exame
Planos para reconstruir o maior avião do mundo são confirmados pela Ucrânia
Eleições nos EUA: milhares de votos ainda devem ser contabilizados; veja a situação
USP aprova novo sistema de ingresso de estudantes via Enem
Mega da Virada sorteia R$ 450 milhões, maior prêmio da história do concurso
```

HTML (*HyperText Markup Language*) é linguagem de marcação de hipertexto padrão para criação de páginas para a internet. Um *script* em *html* é um conjunto de instruções que diz ao seu *navegador web* (o interpretador da linguagem) como deverá ser montada uma página no monitor do seu PC. Sua principal função é marcar e definir a estrutura de uma página na *web*. *Tags* são os marcadores.

Depois, na linha 10, carregamos na variável *'response'* o primeiro item da tupla, o documento *html*, de um dos canais, *cnn*, *veja* ou *yahoo*. Na linha seguinte, convertemos o documento numa estrutura *BeautifulSoup*, com todos os seus objetos identificados e, por isso, facilmente acessíveis.

No laço de repetição *for*, linha 13, cada texto dentro da estrutura *BeautifulSoup* com *tag html 'h3'* para *cnn* e *yahoo*, ou *'h2'* para veja, é capturado e mostrado no terminal do *VSCode*. Assim, teremos uma lista com as manchetes do canal de notícias que quisermos.

Poderíamos automatizar ainda mais nosso *scrapper* incluindo um temporizador que atualizaria a lista toda hora cheia, disparando o programa; ou fazer essa raspagem em mais de um canal e deletando notícias semelhantes. Dá para criar também uma *WordCloud* (nuvem de palavras) com as palavras que mais aparecerem nas manchetes dos principais canais de notícias na *web*. São muitas as possibilidades.

2. Experimentos com páginas *html*

Um *script* para o leitor fazer experimentos com documentos *html* requisitados na *web*, usando a biblioteca *BeautifulSoup*.

```
# pip install bs4
from bs4 import BeautifulSoup

html = """
<html>
   <head>
      <title>
         My Page
      </title>
   </head>
   <body>
      <div class="city">
         <p> My first paragraph. </p> <p> My second paragraph. </p>
      </div>
   </body>
 </html> """
```

soup = BeautifulSoup(html, 'html.parser')

 # mostra todo o documento html
print(soup.prettify())

 # texto na primeira tag 'p', o primeiro paragrafo
print(soup.find('p').text) # My first paragraph.

 # lista com os todos os paragrafos
print(soup.find_all('p')) # [My first paragraph. , My second paragraph.]

 # lista com todas as tags class="city"
print(soup.find_all(class_='city')) # [<div class="city" …]

 # o texto na tag div class="city"
print(soup.find("div", attrs={"class": "city"}).text) # My first paragraph. My second paragraph.

Veja a saída no terminal do *VSCode*:

```
27  # Find by artibute
28  print(soup.find_all(class_='city'))
29
30  # Find by tag and attributes
31  print(soup.find("div", attrs={"class": "city"}).text)
32

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

johnny@johnny-Aspire-E1-571:~/Desktop/ANTENNA/NOV_2022$ /bin/python3
<html>
 <head>
  <title>
   My Page
  </title>
 </head>
 <body>
  <div class="city">
   <p>
    My first paragraph.
   </p>
   <p>
    My second paragraph.
   </p>
  </div>
 </body>
</html>

 My first paragraph.
[<p> My first paragraph. </p>, <p> My second paragraph. </p>]
[<div class="city">
<p> My first paragraph. </p> <p> My second paragraph. </p>
</div>]

 My first paragraph.   My second paragraph.
```

Veja que com *print(soup.find('p').text)* podemos ver o texto do primeiro parágrafo; já *print(soup.find_all('p'))* mostra uma lista com todos os parágrafos do documento *html*.

3. Recortando arquivos de vídeo

Nesse *script* com Python vamos selecionar duas partes de um arquivo de mídia no formato *mp4,* gravá-las separadas no HD, e depois juntar essas duas partes numa só e salvá-la.

Esse é um *script* que pode ser útil quando queremos apagar partes de um arquivo de vídeo *mp4* (ou outro formato suportado). Aqui é prudente fazer uma cópia do arquivo original que vai sofrer os cortes na mesma pasta do *script*. Depois, é só salvar de volta o arquivo final para a pasta do arquivo original com o nome deste.

video_editor_python.py

home > johnny > Desktop > ANTENNA > NOV_2022 > video_editor_python.py > ...

```python
#pip install moviepy
from moviepy.editor import *

path = '/home/johnny/Desktop/ANTENNA/NOV_2022/'

# um objeto clip de um arquivo de video
clip = VideoFileClip(path + 'ocean_with_audio.mp4')

# crie um clip com os primeiros 5 segundos do video original
clip_1 = clip.subclip(0,5)

# e o salve como um arquivo mp4
clip_1.write_videofile(path + 'clip_1.mp4')

''' abra esse arquivo com um duplo clique.'''

# crie outro clip de 5 segundos entre 25 e 30 segundos
clip_2 = clip.subclip(25,30)

# e o salve como um segundo arquivo mp4
clip_2.write_videofile(path + 'clip_2.mp4')

# junte os dois clips criados e o salve numa pasta.
final = concatenate_videoclips([clip_1, clip_2])
final.write_videofile(path + 'ocean_cliped.mp4')

''' abra esse arquivo com um duplo clique.'''
# END
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

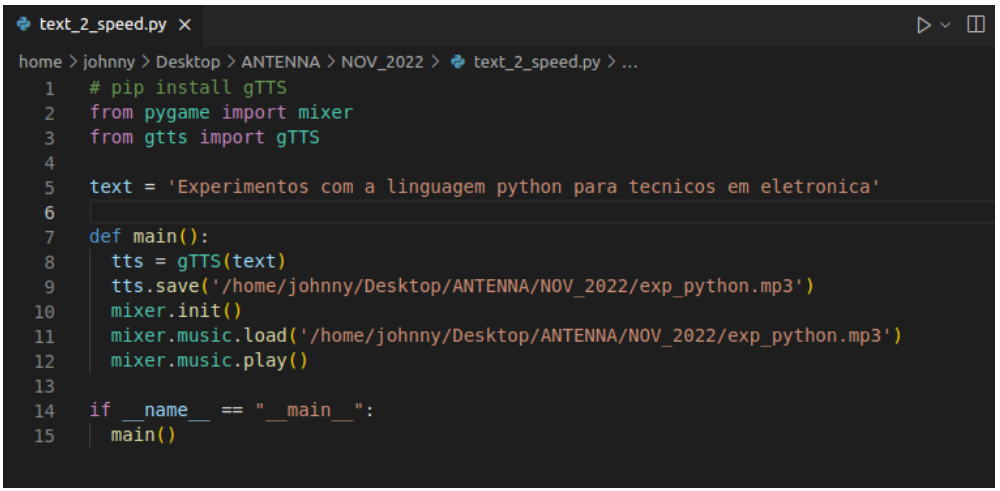
```
Moviepy - Done !
Moviepy - video ready /home/johnny/Desktop/ANTENNA/NOV_2022/clip_1.mp4
Moviepy - Building video /home/johnny/Desktop/ANTENNA/NOV_2022/clip_2.mp4.
MoviePy - Writing audio in clip_2TEMP_MPY_wvf_snd.mp3
MoviePy - Done.
Moviepy - Writing video /home/johnny/Desktop/ANTENNA/NOV_2022/clip_2.mp4
```

Outros métodos da biblioteca *moviepy* que vale a pena testar:

```
# mostra duracao do audio em segundos
audio = clip.audio
print(audio.duration)

# salva arquivo de audio no formato mp3
audio.write_audiofile(path + 'test_audio.mp3')

# acelera o video em 2 vezes e o salva
speedx2 = VideoFileClip(path + clip).fx(vfx.speedx, 2)
speedx2.write_videofile(path + 'clip_X2.mp4')
```

4. Leitor de textos com Python

Um experimento interessante é este pequeno *script* em Python que lê para você com voz robótica um arquivo no formato *.txt* (texto). Precisaremos ter as bibliotecas *'pygame'* e *'gTTS'* instaladas via terminal com *pip install pygame* e *pip install gTTS*. Veja a listagem na tela abaixo.

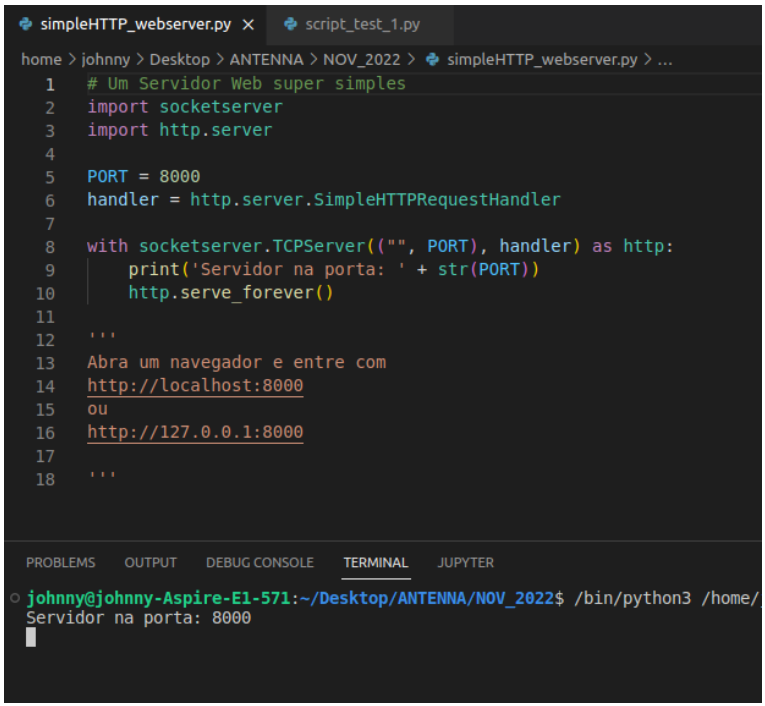


Depois de executado, esse programa cria numa pasta no s*eu PC* um arquivo *.mp3* a partir de um arquivo *.txt*; depois abre esse arquivo de áudio para ser ouvido. Podemos 'ler' um livro digital ouvindo-o.

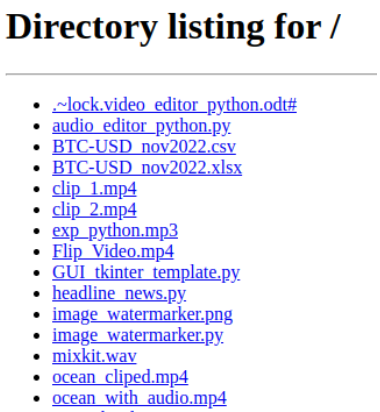
5. Um Servidor Web super simples

O que é em nosso mundo digital um *Servidor*? Um Servidor é um *software* que fica o tempo todo em *standby* dentro de um supercomputador esperando que alguém requisite seus serviços. Um Servidor de *web* quando solicitado serve documentos *html* ao requisitante.

Com somente uma biblioteca e um módulo Python, *'socketserver'* e *'http.server'*, podemos montar em nosso (não super) computador um Servidor de páginas *html*.



Depois de executado, esse *script* cria um Servidor web que pode ser acessado pelo endereço http://localhost:8000, onde aparece a listagem dos arquivos na diretoria atual do seu PC; e que podem ser acessados normalmente clicando sobre seus nomes. Uma vez lançado, qualquer usuário conectado à mesma rede local pode acessar também essa página e a seus documentos.



6. Colocando uma marca d'água numa imagem

Outro *script* bastante interessante com Python, como colar um texto próprio numa imagem, por razões de direitos autorais, por exemplo. Veja a tela abaixo.

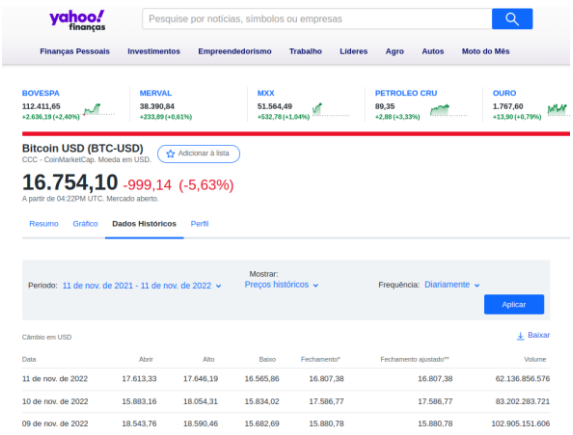
```
home > johnny > Desktop > ANTENNA > NOV_2022 > image_watermarker.py > ...
1  from PIL import Image
2  from PIL import ImageDraw
3
4  def watermark_Image(img_path,output_path, text, pos):
5      img = Image.open(img_path)
6      drawing = ImageDraw.Draw(img)
7      black = (10, 5, 12)
8      drawing.text(pos, text, fill=black)
9      img.show()
10     img.save(output_path)
11
12 path = '/home/johnny/Desktop/ANTENNA/NOV_2022/'
13 img = path + 'yingyang.png'
14 text = 'Experimentos com PYTHON!'
15
16 watermark_Image(img, path + 'watermarked.png',text , pos=(10, 10))
17
```

Aqui importamos dois módulos da biblioteca *PIL.* Nosso programa é uma única função, *watermark_Image()*, que recebe 4 parâmetros: a localização da imagem, o nome e onde vai ser guardada a nova imagem com a marca d'água; o texto e a sua posição na imagem que vamos marcar. Veja o resultado ao lado.

7. Plotando um gráfico simples

Agora, digamos que queremos observar os números e o gráfico do comportamento de um índice econômico qualquer num dado período, como a moeda digital *bitcoin*, por exemplo.

Nossa fonte de dados será um arquivo *.csv* baixado de uma página *web* que forneça esses dados gratuitamente. Aqui estamos acessando a página da *Yahoo!Finanças*. Veja a tela da *yahoo* abaixo.

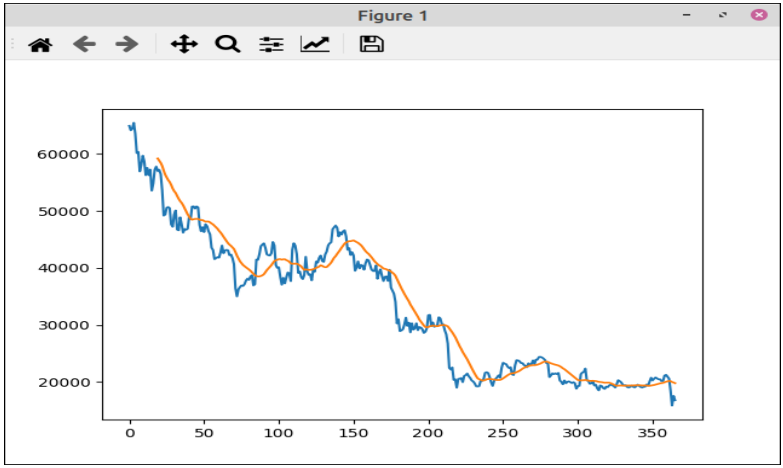


Selecione o período desejado e clique no ícone '*Baixar*', à direita embaixo do botão azul da tela. Escolha uma pasta para salvar o arquivo solicitado.

Agora, copie o *script* a seguir no seu editor *VSCode* e o execute. O terminal vai mostrar os 5 primeiros dados dos preços de fechamento da moeda digital em um ano; e os 5 últimos dados da média móvel de 20 dias dos mesmos preços de fechamento.

Também uma plotagem será gerada com os dados dessas duas bases de dados, atribuídas às variáveis *'btc_usd'* e *'sma_20'*.

```
script_test_1.py
home > johnny > Desktop > ANTENNA > NOV_2022 > script_test_1.py > ...
1   import pandas as pd
2   import matplotlib.pyplot as plt
3   import seaborn as sns
4
5   btc_usd = pd.read_csv('/home/johnny/Desktop/ANTENNA/NOV_2022/BTC-USD_nov2022.csv')
6   sma_20 = btc_usd.rolling(window=20).mean()
7
8   print(btc_usd.head())   # os primeiros 5 elementos do arquivo csv
9   print()
10  print(sma_20.tail())   # os ultimos 5 da media movel de 20 dias
11
12  plt.plot(btc_usd['Close'])
13  plt.plot(sma_20['Close'])
14
15  plt.show()      # plota linhas 'Close' de 2 bases de dados
16

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

johnny@johnny-Aspire-E1-571:~$ /bin/python3 /home/johnny/Desktop/ANTENNA/NOV_2022/script_test_1.py
         Date          Open          High           Low         Close     Adj Close       Volume
0  2021-11-11  64978.890625  65579.015625  64180.488281  64949.960938  64949.960938  35880633236
1  2021-11-12  64863.980469  65460.816406  62333.914063  64155.941406  64155.941406  36084893887
2  2021-11-13  64158.121094  64915.675781  63303.734375  64469.527344  64469.527344  30474228777
3  2021-11-14  64455.371094  65495.179688  63647.808594  65466.839844  65466.839844  25122092191
4  2021-11-15  65521.289063  66281.570313  63548.144531  63557.871094  63557.871094  30558763548

             Open          High           Low         Close     Adj Close        Volume
361  20136.138477  20438.920215  19932.116699  20199.892188  20199.892188  3.973643e+10
362  20199.420703  20504.729785  19855.909570  20169.979004  20169.979004  4.456478e+10
363  20169.704492  20468.492774  19691.471240  20011.331006  20011.331006  4.848534e+10
364  20011.202246  20409.339160  19544.623633  19932.046143  19932.046143  5.102249e+10
365  19933.249609  20329.245313  19416.304395  19812.005811  19812.005811  5.332411e+10
```



Observe no gráfico gerado acima, que o *bitcoin* em um ano caiu bastante e, pela linha laranja da média móvel, há algumas semanas está sem tendência.

8. Um gravador de voz com Python

Por fim, Vamos montar um gravador digital em poucas linhas de código Python; útil quando se deseja criar muitos arquivos pequenos com vozes e tons para automação. Comece instalando as bibliotecas indicadas logo no início do *script*.

home > johnny > Desktop > ANTENNA > NOV_2022 > script_test_2.py > ...

```
# Python Voice Recorder
# pip install wavio
# pip install scipy
# pip install sounddevice

import sounddevice as device
from scipy.io.wavfile import write
import wavio

def Record_Voice(time):
    duration = time
    sample = 44100
    recorded = device.rec(int(duration * sample),
    samplerate=sample, channels=2)
    device.wait()

    wavio.write('/home/johnny/Desktop/ANTENNA/NOV_2022/audio_teste.wav',
    recorded, sample, sampwidth=2)

Record_Voice(10)    # grava por por 10 segundos
```

Depois de executado, o *script* grava 10 segundos de áudio pelo microfone do PC e o salva num arquivo '*audio_teste.wav'* numa pasta no seu PC.

Fim de ano, hora de (mais uma vez) tentar planejar nossa vida; e, para isso, roubamos tempo e, assim, não estaremos presentes na edição de Antenna de dezembro desse ano. Mas voltamos em janeiro de 2023 para aplicar tudo aquilo que vimos nos 12 primeiros artigos num novo desafio: como controlar hardware com Python. Até lá!

LINKS:

Arquivos mp4 de graça:
https://www.sample-videos.com/

Histórico de índices econômicos
https://br.financas.yahoo.com/quote/BTC-USD/history?p=BTC-USD

Loudness, Uma História...
Parte IV

O CN-100

Álvaro Neiva*

Numa edição passada (out/2021), chegamos ao circuito final do equalizador de loudness ativo e o incluímos como parte de um pré-amplificador com controle de tonalidade para sinais com nível de linha (1Vrms). O Renato Colicigno criou uma PCB e foi montado um protótipo para aferição do resultado.

O motor desse desenvolvimento foi trazer para a realidade um projeto anterior, que havia batizado de AN-1, o qual seria um pré-amplificador de alto desempenho, com funcionalidades presentes em equipamentos comerciais de alto nível, mas passível de montagem caseira (DIY), fruto das conversas de um grupo de interessados, conhecidos através das redes sociais. Nesse grupo, um incentivador constante foi o Gustavo Chin, junto com o Renato Colicigno, o Miguel Nabuco e outros, como o João Yazbek, que chamou a atenção para usar o controle de volume ativo, por exemplo, que trouxeram inúmeras contribuições para o desenvolvimento do projeto, isso aí por 2015...

O projeto ficou pronto, com entradas e saídas balanceadas e desbalanceadas, seleção de entradas por relés, ajuste de nível no painel traseiro para cada entrada, controle de tonalidade de duas bandas com um opamp por banda, potenciômetros de baixo valor para mínimo ruído, amplificador de fones classe A push-pull (o pequeno notável...), saída e entrada para gravador com buffer e loop para processador externo.

Bom, ficou complexo e caro... isso esfriou essa história, mesmo que eu demonstrasse o amplificador de fones funcionando... afinal, todos tinham seus afazeres profissionais e desenvolvimento de produto toma tempo e recursos...

Ao publicar o circuito de loudness em outubro de 2021, o Renato levantou a ideia de fazer uma versão simplificada do projeto, passível de usar num amplificador integrado ou de forma separada.

Em homenagem ao Gustavo, acrescentamos a letra C, ficando o novo projeto batizado de **CN 100**. A segunda montagem (com entradas e saídas balanceadas), aparece na figura 1.

***Engenheiro Eletricista**