

PYTHON

[50 Lições]

"Educação é algo importante demais para ser mantida nas mãos do governo".

Professor João Antonio

www.professorjoaoantonio.com



O que te Espera?

Lição 1.	Introdução a Python	4
Lição 2.	O Editor de Código	7
Lição 3.	A Função print()	11
Lição 4.	Tipos de Dados no Python	16
Lição 5.	Cálculos e Mais Cálculos	21
Lição 6.	Variáveis.....	25
Lição 7.	Alguns Detalhes a Mais	31
Lição 8.	Valores Booleanos [Falso / Verdadeiro]	34
Lição 9.	Funções Úteis	37
Lição 10.	Recebendo Dados do Usuário	40
Lição 11.	Listas	43
Lição 12.	Vamos Exercitar Muito?	48
Lição 13.	Mais Sobre Listas	51
Lição 14.	A Função range().....	54
Lição 15.	A Estrutura de Repetição for.....	56
Lição 16.	A Estrutura de Repetição while.....	59
Lição 17.	Estrutura Condicional if.....	63
Lição 18.	Caracteres Especiais nas Strings	66
Lição 19.	Formatando Texto.....	70
Lição 20.	Muitos Programas para Estudar!	73
Lição 21.	Construindo Nossas Funções	77
Lição 22.	As Variáveis Dentro das Funções	81
Lição 23.	Funções Retornando Valores	84
Lição 24.	List Comprehensions.....	87

Lição 25.	Mais Segredos com while	90
Lição 26.	Brincando mais com Listas e Strings	94
Lição 27.	Tuplas	98
Lição 28.	Semelhanças entre Tuplas e Listas	101
Lição 29.	Dictionaries [Dicionários].....	105

Lição 1. Introdução a Python

1.1. O que é Python?

Python é uma linguagem de programação de alto nível.

Linguagem de Programação: conjunto de códigos, palavras especiais e regras que nos permite criar programas de computador [aplicativos e sites da internet, por exemplo].

Alto nível: significa que ela é bem próxima da nossa linguagem [da linguagem humana], portanto, é mais agradável de aprender e entender!

Python é usada para muitas coisas, como programação de sites [ou seja, fazer páginas da web], análise de dados, computação científica e inteligência artificial.

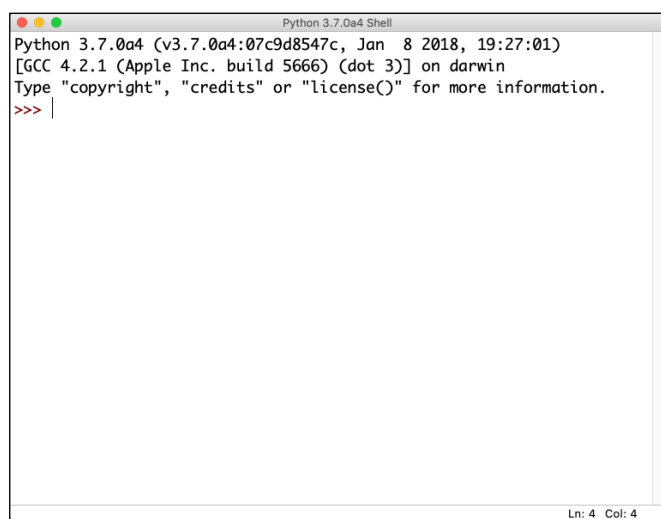
Python é **muito fácil de aprender** e muito poderosa. Usaremos aqui a **versão 3.7**.

Como se "fala" o nome Python, João?

Que bom que perguntou, caro leitor, amiga leitora. O nome da linguagem se fala normalmente "**Páiton**" ou "Páisson" [esse "ss" é falado com a língua entre os dentes se você quiser "dar uma de bonzão, diferentão, f*dão que fala inglês com sotaque perfeito"].

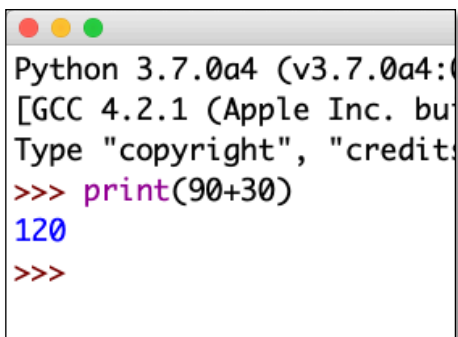
1.2. Como é o Python?

O programa onde podemos treinar e usar a linguagem Python é chamado de IDLE [que é a sigla em inglês para **Ambiente de Aprendizado e Desenvolvimento Integrado**]. Ele é composto por uma janela principal chamada **Shell**.



A janela do Shell do IDLE do Python sendo mostrada ao lado. Esta é a primeira coisa que você verá quando abrir o programa do Python.

A parte mais importante desta janela é chamada de **Prompt** [ou "**aviso**"]. É justamente aquele sinal de três "setas" [**>>>**] que aparece para esperar os seus comandos.



```
Python 3.7.0a4 (v3.7.0a4:0)
[GCC 4.2.1 (Apple Inc. bu
Type "copyright", "credits
>>> print(90+30)
120
>>>
```

Neste exemplo, o usuário digitou `print(90+30)` e pressionou ENTER. O Shell respondeu mostrando o resultado da operação matemática – o 120 – e depois mostrou novamente o Prompt para esperar o próximo comando.

Então, você percebeu que no Shell é assim:

- [a] O Shell do Python mostra o prompt pra você.
- [b] Você digita um comando e pressiona ENTER;
- [c] O Shell do Python obedece ao seu comando;
- [d] O Shell do Python mostra o prompt novamente, esperando o seu próximo comando.

Ou seja, o shell do Python é um ambiente **Interativo**, porque ele interage com você, usuário.

E por que esse tal de "print()" que você escreveu aí, João? Eu não poderia simplesmente escrever `90+30` no prompt?

Sim, meu amigo leitor. Sim, minha amiga leitora. No prompt, você pode escrever diretamente a expressão matemática desejada e o shell vai responder normalmente. Veja na imagem abaixo:



```
Python 3.7.0a4 Shell
>>> print("NOVO")
NOVO
>>> "Não existe almoço grátis"
'Não existe almoço grátis'
>>> 'Leia Mises'
'Leia Mises'
>>> print("Leia Rothbard")
Leia Rothbard
>>> print('Leia Hayek')
Leia Hayek
>>> 12+18
30
>>> print(800)
800
>>> 950
950
>>> print(340-230)
110
>>>
```

Veja algumas das formas de você solicitar que o shell do Python mostre dados.

Vamos explicar essa imagem bem claramente agora:

- [a] **Não é necessário** usar a função `print` para mostrar resultados – você pode escrever diretamente o que deseja e o shell vai mostrar.

- [b] Se você pede que o shell mostre textos, **eles devem ser escritos entre aspas** [simples ou duplas, tanto faz];
- [c] Se você pede que o shell mostre números ou resultados de cálculos, **eles não podem ser escritos entre aspas** [pois seriam entendidos como textos].

1.3. Vamos Exercitar sobre a Lição 1

1.3.1. Usando o shell do Python, escreva comandos para mostrar

```
O seu nome
>>>
A sua idade
>>>
O resultado da soma da sua idade com o número 30
>>>
O resultado da soma da sua idade com o resultado do comando anterior
>>>
O nome da cidade onde você nasceu
>>>
A data em que você está fazendo esta atividade
>>>
```

1.3.2. O que vai ser mostrado no shell do Python para cada comando a seguir:

```
>>> print("João Antonio")

>>> print(80 + 90)

>>> 8+10

>>> print("30+20")
```

Telegram: Bora trocar ideias no grupo do Telegram? Meu username é: *@ProfessorJoaoAntonio*.

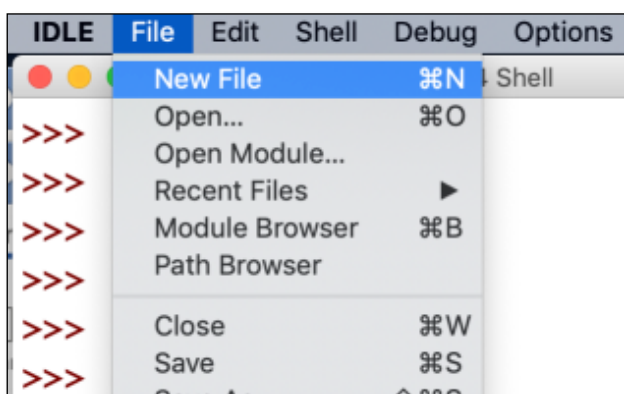
Lição 2. O Editor de Código

2.1. Conhecendo o Editor de Código

O shell é a janela principal do IDLE do Python [isso você já sabia porque aprendemos na Lição 1].

Mas o shell não é a única janela do programa. Existe uma outra janela que usaremos bastante chamada **Editor de Código** [ou simplesmente **Editor**].

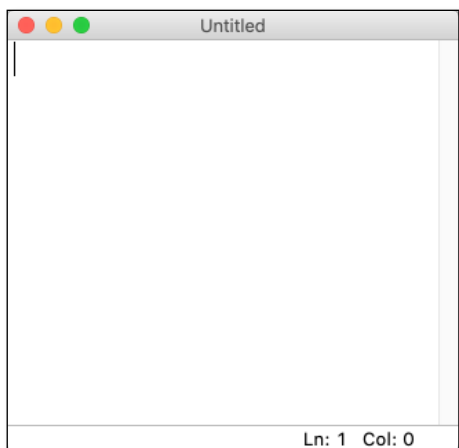
Você pode abrir várias janelas do Editor enquanto estiver trabalhando no IDLE. Para abrir uma nova janela do Editor do Python, você aciona o menu **File** [ou **Arquivo**, no menu superior] e, dentro dele, aciona **New File** [**Novo Arquivo**, em português].



Para abrir uma nova janela do Editor, acione a opção New File que aparece dentro do menu File.

CTRL+N é a tecla de atalho que aciona essa ação

O Editor parece uma coisa bem sem-graça... é uma janela toda em branco, apenas com o **cursor** piscando... pronta para você digitar lá dentro.



Um exemplo de janela do Editor. A palavra Untitled ["Sem Título" traduzido para o português] lá em cima é o nome que se dá quando o conteúdo da janela ainda não foi salvo.

O Editor não é interativo como o Shell. O Editor **não obedece** a você. Ele **não responde imediatamente** aos seus comandos. Você vai escrever no Editor os seus programas – os seus scripts.

Programa: *é uma sequência de comandos que o programador escreve e que o computador obedece exatamente em ordem.*

No Shell, cada comando é obedecido quando você pressiona ENTER – um por vez – manualmente. Num programa, você escreve todos eles e depois manda que sejam obedecidos de uma vez.

A imagem a seguir mostra um exemplo de um código escrito na linguagem Python dentro na janela do Editor. Perceba que o editor é "esperto" e consegue colorir certas palavras e expressões especiais com cores distintas. Você vai se acostumar com isso.



```
# Exemplo

if "imposto" == roubo:
    while vivo:
        a = "so"
        a += "ne"
        print(a + "gue")
estado.legitimo = False
```

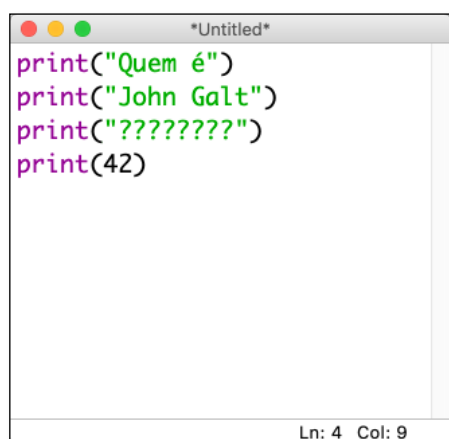
Pequeno código de programação em Python escrito na janela do Editor. Note a presença da função "print" lá embaixo. Sim, aqui no Editor nós a usaremos muito!

*Note lá em cima a palavra "*Untitled*" entre asteriscos. Isso indica que o conteúdo da janela foi modificado, mas ainda não foi salvo.*

NUNCA SE ESQUEÇA DE SALVAR

2.2. Nosso Primeiro Código

Para começar nossa aventura com o Editor, vamos escrever algumas linhas de comando usando a função print. Que tal essas aqui:



```
print("Quem é")
print("John Galt")
print("????????")
print(42)
```

Nesta janela do Editor, escrevemos 4 comandos usando a função print.

Cada um deles é escrito em uma linha diferente. É só pressionar ENTER para descer para a próxima linha.

As cores foram dadas pelo próprio editor [esperto, não?]

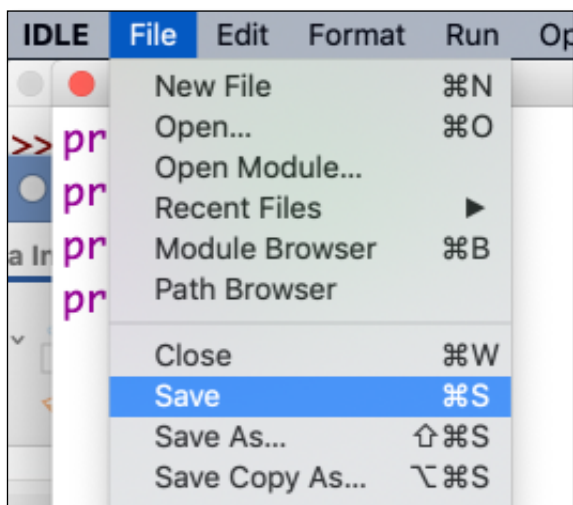
O próximo passo é salvar o nosso código [para poder usarmos depois, se quisermos].

Ei, João, eu queria ver esse código sendo obedecido pelo computador! Como eu faço isso? Como se chama essa ação?

Amigo leitor, amiga leitora, a ação de **fazer o computador obedecer ao nosso código** é chamada de **executar o programa** ou **rodar o programa** ["rodar" porque os programas antigamente eram gravados em fitas, que giravam, giravam e giravam... você não é dessa época e isso é só uma curiosidade].

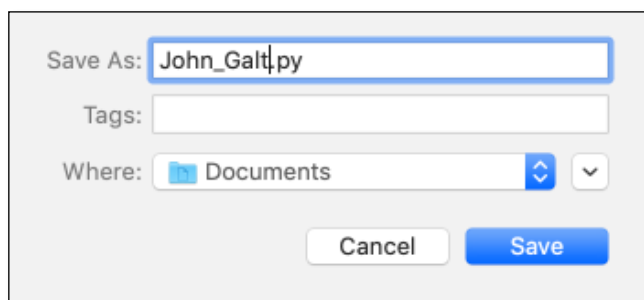
Antes, porém, de executar o programa para ver o computador obedecendo a ele, **temos que salvar o nosso código** [o IDLE do Python não deixa você rodar o programa sem ter salvo... e você vai perceber isso da pior maneira: vai ficar de saco cheio dos lembretes!].

Acione, lá em cima, no menu superior, a opção **File [Arquivo]** e clique em **Save [Salvar]**. É possível usar, também, o atalho **CTRL+S**.



Comando File / Save [Arquivo / Salvar], que também pode ser acionado pela combinação de teclas CTRL+S.

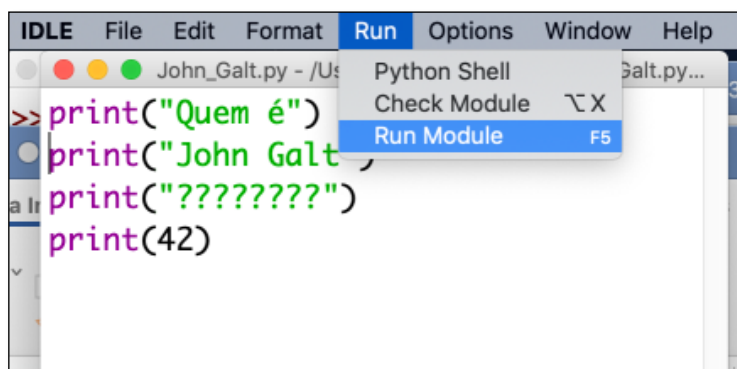
Logo em seguida, você terá de colocar um nome do arquivo que está criando. Esse arquivo terá o nome que você disser seguido da **extensão .py**.



Salvando o nosso código como John_Galt.py

Depois disso, ele poderá ser normalmente executado [ou seja, veremos ele funcionando]

Com o código escrito, arquivo salvo, churrasco no fogo e Roupas Nova tocando na vitrola, já podemos executar o programa – ou seja, fazê-lo funcionar. Basta acionar o comando **Run Module [Executar Módulo]** que se encontra dentro do menu superior **Run [Executar]**.



Comando Run Module, dentro do menu Run.

Ele pode ser acionado também pela tecla F5.

Todo programa é executado [ou seja, é obedecido] numa janela do Shell. Programas são construídos na janela do Editor, mas são executados na janela do Shell. Olha só o resultado deste nosso exemplo na figura a seguir:



```
>>>
==== RESTART: /Users/joao/Documents/John_Galt.py =====
Quem é
John Galt
????????
42
>>>
```

Janela do Shell do Python mostrando o resultado do programa que executamos. Cada comando print que foi usado escreve sua mensagem em uma linha diferente.

Então é isso! Por hoje é só revisar para você não esquecer que:

- [a] Você escreve os programas em Python numa Janela do Editor.
- [b] Você precisa salvar o documento [código] Python num arquivo com extensão .py.
- [c] Quando você executa o código [manda ele "trabalhar"], ele é executado [mostra seus resultados] numa janela do Shell do IDLE Python.

2.3. Vamos Exercitar acerca da Lição 2

2.3.1. Quais os resultados da execução dos códigos a seguir?

```
print("Qual a resposta para a vida, o universo e tudo mais?")
print(20+22)
print("É Sério! Nós calculamos!")
```

```
print("Qual a resposta do cálculo abaixo:")
print("80 + 12")
```

Você tem Instagram? Me segue lá: [*@ProfessorJoaoAntonio*](#).

Lição 3. A Função print()

3.1. Vamos nos conhecer melhor?

Desde a primeira lição, estamos usando a função print(). Ela é simplesmente **a função que faz a saída dos dados na tela**.

Essa é a explicação que você verá em todos os livros de Python: **"a função que faz a saída dos dados"**.

Eu, porém, acho melhor explicar [como sempre fiz] da maneira mais clara e mais "para crianças" possível. Espero que você não se sinta constrangido com isso.

Em primeiro lugar, é bom explicar que **você não é o usuário!** Você é o programador. **Você é O CARA!** Você é o cara que está criando um programa que será usado por um usuário.

***Usuário:** é a criatura nefasta que Deus colocou na tua vida para expiar os teus pecados!*

Piadinhas internas, sabe qual é a diferença entre um vírus e um usuário? **O Vírus funciona!**

Bom, acontece que o seu trabalho é **criar um programa** – um aplicativo, talvez – que será usado por um ou mais usuários! E em muitos momentos o seu programa terá que "mostrar" coisas ao usuário, como mensagens, nomes, números, resultados de cálculos, entre outras coisas...

É aí que a função print() atua!

É a função print() que usamos para colocar dados, mensagens, avisos, números e tudo mais na tela. Aquilo que o usuário vai ver quando estiver usando o programa é resultado de uma função print().

Ô, João, por que ficar chamando de "Função"? O que é uma Função? Não posso chamar de "comando print()?"

Poder, até pode... quem sou eu pra te impedir, né? Mas o nome certo é função, mesmo! Print() é uma função na linguagem Python. Veremos isso mais tarde!

3.2. Como Usar a Função print()?

Acredito que você já esteja cansado de usá-la e que já sabe "de cor" [decorado] como usá-la, mas vamos aqui fazer uma lista bem objetiva e, acredito, completa.

De uma forma bem simples, a função print() é usada assim: **print(aquilo que você quer mostrar)**.

3.2.1. Usando print() com textos

Para mostrar textos na tela para o usuário, é só colocar o texto desejado dentro dos parênteses da função **entre aspas** [simples ou duplas] e ele será colocado exatamente daquele jeito na tela [as aspas não aparecerão para o usuário – elas indicam apenas que aquilo é texto].

```
Python 3.7.0a4 Shell
>>> print("Thomas Sowell")
Thomas Sowell
>>> print("Hans Herman-Hoppe")
Hans Herman-Hoppe
>>> print("Murray Rothbard")
Murray Rothbard
>>>
```

Exemplos de uso da função print() com textos no Shell do Python. O resultado é o mesmo se a função for usada numa janela do Editor.

3.2.2. Usando print() com Números e Cálculos

A função print() pode ser usada com números e cálculos [expressões matemáticas] bastando informar, dentro dos parênteses, o que deseja mostrar, **sem aspas**.

```
Python 3.7.0a4 Shell
>>> print(30)
30
>>> print(30+40)
70
>>> print(23-10)
13
>>> print(12*4)
48
>>> print(8/2)
4.0
>>> print(2**4)
16
>>>
```

Note, ao lado, alguns exemplos de impressão [saída] de números e resultados de cálculos simples.

Lembre-se de que no Shell diretamente não é preciso usar a função print() para mostrar resultados, mas estamos nos acostumando a usá-la porque no Editor, ela é obrigatória.

Podemos aproveitar o momento de apresentações e já mostrar alguns dos principais cálculos que podemos fazer no Python:

- [a] Para **somar**, use o símbolo de + [mais], como em **print(80+30)** [isso dá 110]
- [b] Para **subtrair**, use o símbolo de - [traço, ou "menos"], como em **print(20-12)** [isso dá 8]
- [c] Para **multiplicar**, use o símbolo de * [asterisco], como em **print(4*3)** [isso dá 12]
- [d] Para **dividir**, use o símbolo de / [barra], como em **print(18/6)** [isso dá 3.0]
- [e] Para elevar um número a uma **potência**, use ** [duplo asterisco], como em **print(2**3)** [isso é o mesmo que fazer, no caderno, 2^3 – ou seja, dá 8]

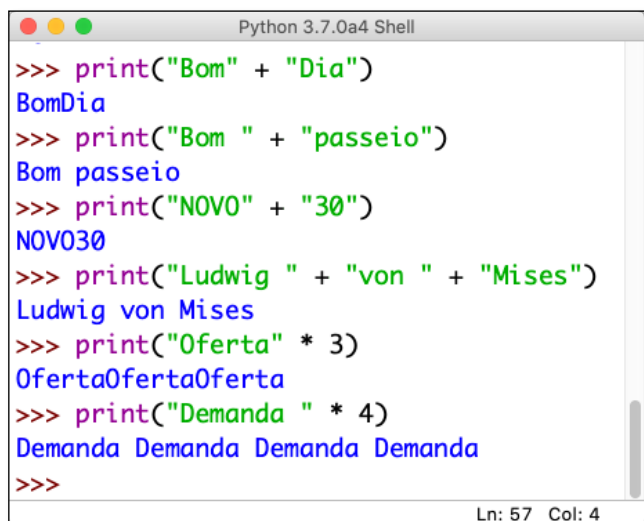
Nas próximas lições, vamos ser apresentados a uma variedade maior de símbolos que fazem cálculos [esses símbolos são chamados de **operadores** matemáticos].

E só mais um lembrete importante: não importa se há espaços entre os símbolos num cálculo. Ou seja, se você escreve **3+4** ou **3 + 4** ou **3 + 4**, não faz diferença – todos darão o mesmo resultado.

3.2.3. Usando print() com "Cálculos em textos"

Muita gente pensa que não dá para fazer "cálculos" com textos... mas dá! No Python, especialmente, podemos usar dois operadores de cálculos com textos: o **sinal de +** e o **sinal de *** .

Ou seja, podemos "somar" e "multiplicar" textos. Que tal? Dá uma olhada na próxima imagem:



```
>>> print("Bom" + "Dia")
BomDia
>>> print("Bom " + "passeio")
Bom passeio
>>> print("NOVO" + "30")
NOVO30
>>> print("Ludwig " + "von " + "Mises")
Ludwig von Mises
>>> print("Oferta" * 3)
OfertaOfertaOferta
>>> print("Demanda " * 4)
Demanda Demanda Demanda Demanda
>>>
```

Note o efeito do uso dos sinais de adição [+] e multiplicação [] em textos.*

Você vai usar isso mais do que imagina!

Explicando:

Quando aplicado a textos, o **sinal de + [adição]** passa a realizar uma **concatenação** – ou seja, ele concatena textos.

Concatenar: *é o mesmo que fundir, ou unir, textos. Transforma dois ou mais textos em um só – "tudo junto e misturado".*

Lembre-se que o resultado da concatenação é "tudo junto" mesmo! Note que o comando **print("Bom"+"Dia")** resultou em **BomDia** [sem espaço entre as palavras]. Se você quer colocar um espaço para a coisa ficar organizada, coloque-o dentro das aspas, depois de "Bom " ou antes de "Dia".

[fiz isso lá em cima, na figura, em "Bom " + "passeio"]

Ao usar o operador de **multiplicação * [asterisco]** , relacionando um número a um texto, você simplesmente dá ordem para o Python repetir a exibição daquele texto.

Ou seja, o comando **print("Então é Natal... " * 9)** vai exibir:

Então é Natal... Então é Natal... Então é Natal... Então é Natal... Então é Natal... Então é Natal... Então é Natal... Então é Natal... Então é Natal...

[Ninguém merece... na voz da cantora Simone... não tem como não lembrar]

Aqui vai um Alerta: *quando usados em textos, há certas exigências para os operadores + e *:*

[a] O operador de * [multiplicação] só pode ser usado entre um texto e um número, nunca entre dois textos.

[b] O operador de + [concatenação] só pode ser usado entre dois textos, nunca entre um texto e um número.

[você vai perceber isso quando começar a receber as mensagens de erro caso tente desrespeitar essas regras]

3.3. Função print() com mais de um Argumento

Nós já sabemos que print() é uma função [mais adiante mostraremos o que isso significa]. Nós também sabemos que print() serve justamente para a saída de dados [ou seja, para mostrar na tela, para o usuário, aquilo que estiver dentro dos seus parênteses].

É justamente sobre esses dados que eu quero falar agora: tudo aquilo que estiver dentro dos parênteses de uma função é chamado de **argumento** da função. Por exemplo:

```
print("Não existe almoço grátis")
```

A frase "Não existe almoço grátis" é o **argumento** da função print() mostrada acima.

Já vimos que textos, números e cálculos podem ser argumentos da função print(). Textos devem vir entre aspas e os demais tipos não podem vir entre aspas.

A função print() pode ser escrita com mais de um argumento ao mesmo tempo, basta apresentá-los dentro da função separados por **vírgulas**. Veja o exemplo:

```
print("Número", 30, 40 + 80, "Hoje" * 3)
```

Neste código, a função print() foi usada para mostrar 4 diferentes dados na mesma linha para o usuário separados por um espaço. O resultado deste código será:

```
Número 30 120 HojeHojeHoje
```

Note dois detalhes importantes quando usamos a função print() com mais de um argumento:

- [a] Os argumentos são sempre separados automaticamente por um espaço.
- [b] Os argumentos são apresentados **numa única linha** [diferentemente de quando você usa mais de uma função print() – em que cada função apresentará seu argumento em uma linha diferente].

Olha o exemplo de usar funções print() diferentes:

```
print("Número")  
print(30)  
print(40 + 80)  
print("Hoje" * 3)
```

Isso vai dar o seguinte resultado:

```
Número  
30  
120  
HojeHojeHoje
```

3.4. *Vamos Exercitar a Lição 3*

3.4.1. O que será mostrado na tela nos comandos a seguir?

```
>>> print("Naruto " + "é melhor que " + "Dragon Ball")
```

```
>>> print(7* 3)
```

```
>>> print("7" * 3)
```

```
>>> print(7 * "3")
```

```
>>> print("Casa" * 4)
```

```
>>> print("Casa" + "4")
```

```
>>> print("Casa" * "4")
```

```
>>> print("Casa" + 4)
```

```
>>> print("Casa", 4)
```

```
>>> print("Casa",4 * "Casa")
```

Lição 4. Tipos de Dados no Python

4.1. Por que "Tipos" de Dados?

Tudo que você manipula no Python [textos e números] é considerado **dado**.

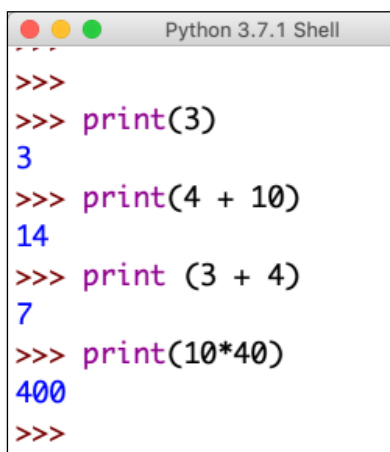
- [a] A letra "a" é um dado.
- [b] O número 30 é um dado.
- [c] A frase "vai faltar areia" é um dado.
- [d] A Indicação de que algo é True [verdadeiro] ou False [falso] é um dado também.

O Python, assim como toda linguagem de programação, aceita **apenas certos tipos de dados** que precisamos conhecer:

4.1.1. Integer [Inteiro]

Quando usamos números "não quebrados", ou seja, números inteiros, o Python os registra como pertencentes ao tipo **integer** [ou **int**, se você for mais íntimo].

Por padrão, todos os números que você utiliza no Python são do tipo integer [exceto, claro, aqueles números que possuem casas decimais – os números "não-inteiros"]. Veja um exemplo de manipulação de números do tipo integer:



```
Python 3.7.1 Shell
>>>
>>> print(3)
3
>>> print(4 + 10)
14
>>> print (3 + 4)
7
>>> print(10*40)
400
>>>
```

Ao lado, alguns comandos print() com números do tipo integer.

Foram realizadas, inclusive, algumas operações matemáticas com eles.

É bom lembrar de alguns detalhes:

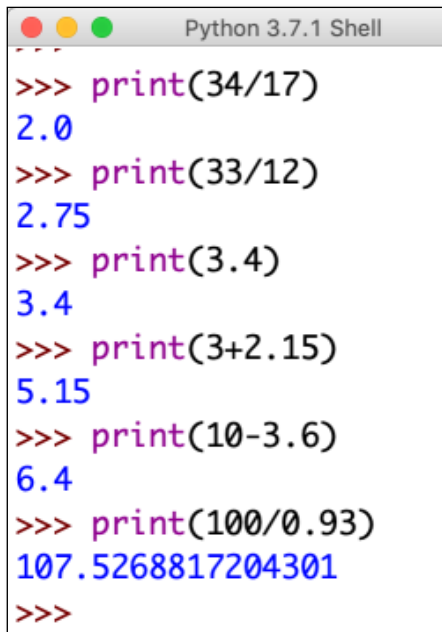
- [a] Somar dois números integer resultará num número integer;
- [b] Subtrair ou Multiplicar números integer também resultará num integer;
- [c] Elevar um Integer a uma potência integer também gerará um integer como resposta.
- [d] Apenas o resultado de uma divisão não num integer! NUNCA!

Para entender essa letra [d], precisamos conhecer um outro tipo de dado:

4.1.2. Float [Flutuante – Número não-inteiro]

Um número do tipo **float** é um número que está preparado para armazenar valores não-inteiros, ou seja, um número que sempre tem casa decimal.

Quando você divide dois números no Python, o resultado sempre será um float. Veja o exemplo abaixo:



```
Python 3.7.1 Shell
>>> print(34/17)
2.0
>>> print(33/12)
2.75
>>> print(3.4)
3.4
>>> print(3+2.15)
5.15
>>> print(10-3.6)
6.4
>>> print(100/0.93)
107.5268817204301
>>>
```

Ao lado, algumas operações com números float. Note que o Python sempre coloca casas decimais [separadas da parte inteira por um ponto].

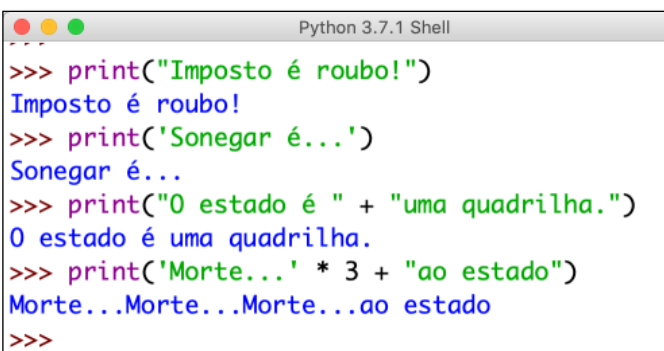
Mesmo que você divida dois números integer, o resultado é necessariamente um float.

Note, também, que qualquer operação [soma, subtração, multiplicação] que envolva números float sempre resultará num número float.

Então, para reconhecer: números float mostram casas decimais separadas da parte inteira por um ponto. Números integer não mostram. Fácil, né? Ou seja, **2.0 é um float. 2 é um integer.**

4.1.3. String ["corda" de caracteres – ou seja, Texto]

O tipo **string** é o mais usado. **Dados do tipo string são textos.** Qualquer dado que você escreva entre aspas duplas ou aspas simples é, automaticamente, classificado como uma string.



```
Python 3.7.1 Shell
>>> print("Imposto é roubo!")
Imposto é roubo!
>>> print('Sonegar é...')
Sonegar é...
>>> print("0 estado é " + "uma quadrilha.")
0 estado é uma quadrilha.
>>> print('Morte...' * 3 + "ao estado")
Morte...Morte...Morte...ao estado
>>>
```

Na janela ao lado, a manipulação de alguns dados do tipo String.

*Usamos, inclusive, os operadores + e * para fazer concatenação e repetição de strings.*

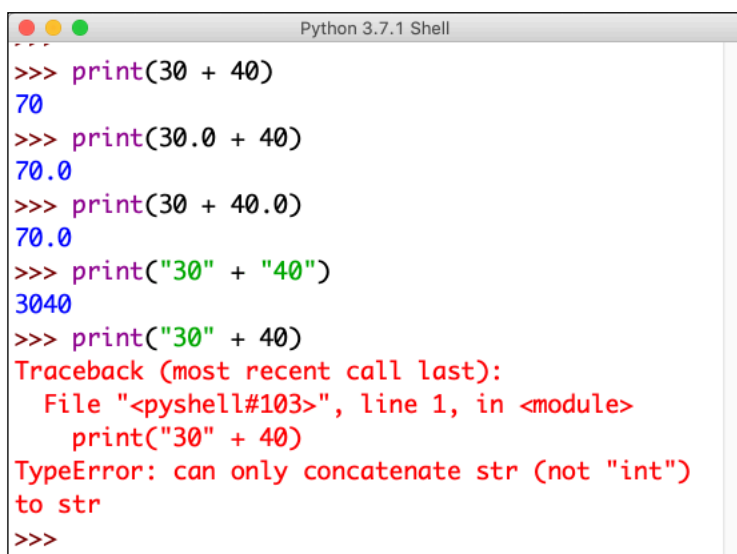
Chegamos num ponto em que dá para explicar melhor o uso dos operadores matemáticos com os tipos de dados.

4.1.4. Operadores e Seus Segredos

Operador + [Sinal de "mais"]

Este operador é usado para **somar** ou **concatenar**.

- [a] Se ele for usado entre **dois números integer**, resultará na **soma em formato integer**.
- [b] Se ele for usado entre **dois números em que pelo menos um deles é float**, o resultado será a **soma dos dois em formato float**.
- [c] Se ele for usado entre **dois dados do tipo string**, o resultado é a **concatenação em um novo string**.
- [d] Se tentar usá-lo **entre uma string e um número qualquer**, dará erro.



```
>>> print(30 + 40)
70
>>> print(30.0 + 40)
70.0
>>> print(30 + 40.0)
70.0
>>> print("30" + "40")
3040
>>> print("30" + 40)
Traceback (most recent call last):
  File "<pyshell#103>", line 1, in <module>
    print("30" + 40)
TypeError: can only concatenate str (not "int")
to str
>>>
```

Alguns exemplos de operações feitas com o operador +.

Note o resultado de cada uma delas e compare com as informações apresentadas acima.

Perceba, também, onde o erro ocorreu.

Operador * [Asterisco]

Este operador pode ser usado para **multiplicar** e para **repetir**.

- [a] Se ele for usado entre **dois números integer**, resultará no **produto [multiplicação] dos dois em formato integer**.
- [b] Se ele for usado entre **dois números em que pelo menos um deles é float**, o resultado será o **produto [multiplicação] dos dois em formato float**.
- [c] Se tentar usá-lo **entre uma string e um número integer**, será gerada uma string contendo a **repetição da string original** tantas vezes quanto for o valor do número.
- [d] Se ele for usado **entre uma string e um número float**, ocorrerá **um erro**.
- [e] Se ele for usado entre **dois dados do tipo string**, ocorrerá **um erro**.

Podemos ver exemplos do uso do * na imagem a seguir:

```

Python 3.7.1 Shell
>>> print(23 * 4)
92
>>> print(23 * 4.0)
92.0
>>> print(23.0 * 4)
92.0
>>> print("Mises " * 5)
Mises Mises Mises Mises Mises
>>> print("Hayek " * 3.0)
Traceback (most recent call last):
  File "<pyshell#168>", line 1, in <module>
    print("Hayek " * 3.0)
TypeError: can't multiply sequence by non-int of
type 'float'
>>> print("Keynes " * "Hayek")
Traceback (most recent call last):
  File "<pyshell#169>", line 1, in <module>
    print("Keynes " * "Hayek")
TypeError: can't multiply sequence by non-int of
type 'str'
>>>

```

*Exemplos de utilização do operador *:*

Note os resultados das multiplicações com floats e integers. E, claro, os erros no final do exemplo.

Operadores – [sinal de "menos"] e [/barra]

Esses dois operadores **só podem ser usados entre dois números** [integer ou float, não importa]. Não é possível usar qualquer um desses dois operadores com strings [ou seja, dá erro].

Quanto ao **operador –** [subtração], temos que:

- [a] Se ele for usado **entre dois números integer**, o número resultante **será integer**.
- [b] Se ele for usado **com pelo menos um float**, o número resultante **será float**.

Quanto ao **operador /** [divisão], temos a seguinte condição:

- [c] Qualquer utilização deste operador gerará uma divisão entre dois números e o resultado **sempre será um float**.

```

Python 3.7.1 Shell
>>> print(34-12)
22
>>> print(34.0-12)
22.0
>>> print(34-12.0)
22.0
>>> print(34/17)
2.0
>>> print(34/17.0)
2.0
>>> print("Teste"/2)
Traceback (most recent call last):
  File "<pyshell#149>", line 1, in <module>
    print("Teste"/2)
TypeError: unsupported operand type(s) for /: 'str' and 'int'
>>>

```

Exemplo do uso dos operadores – [subtração] e / [divisão].

Perceba os resultados.

Esses dois operadores não podem ser usados com textos.

4.2. Vamos Exercitar sobre a Lição 4

4.2.1. Indique quais os tipos dos dados a seguir:

"Não existe almoço grátis"

18

18.0

"18"

'18+12'

18+12

18/3

18*3

"Oba " * 3

"Oba " + "Legal"

'Oba' + ' Legal'

18 + 9.0

Lição 5. Cálculos e Mais Cálculos

5.1. Revendo os Operadores Aritméticos

Na lição passada, fomos apresentados a alguns dos principais operadores aritméticos usados na linguagem Python. Vamos revê-los rapidamente agora:

O **operador +** é usado para somar dois números. Portanto

```
print(80+50)
```

vai apresentar na tela o **valor 130 [integer]**.

O **operador -** é usado para subtrair o segundo número de um primeiro número, portanto o trecho

```
print(90-30)
```

vai resultar no aparecimento do **número 60 [integer]**.

Se você utilizar o **operador *** com dois números, você os multiplica. Logo, o código

```
print(30*4)
```

vai mostrar na tela o **número 120 [integer]**.

Por fim, se você utiliza o **operador /**, você divide o primeiro número pelo segundo. Portanto,

```
print(20/5)
```

vai apresentar, na tela, o **número 4.0 [float]**. [toda operação de divisão resulta num número **float**]

Se você utilizar o **operador ** [asterisco duplo]**, você estará elevando o primeiro número à potência do segundo número. Isso significa que o código

```
print(3**2)
```

vai apresentar o **número 9 [integer]** como resultado na tela. [afinal, **3² é 9**].

Todos esses já havíamos visto. Vamos a dois outros operadores relacionados à divisão entre dois números:

O **operador // [barra dupla]** é usado para obter o **quociente de uma divisão inteira** [integer].

O **operador % [sinal de porcentagem]** é usado para obter o **resto de uma divisão inteira**. [integer]

O que são essas duas? Ahhh... você lembra da escola? Quando a gente aprendeu a calcular uma divisão com resto? Era aquele tipo de cálculo que aqui chamamos de "divisão inteira" – uma divisão que usa apenas números inteiros.

Dá uma olhada na imagem a seguir, em que divido 14 por 3, obtendo 4 com resto 2:

$$\begin{array}{r}
 14 \overline{) 3} \\
 \underline{4} \\
 \text{resto: } 2
 \end{array}
 \quad \leftarrow \text{resultado do cálculo } 14 // 3$$

resto: 2 \leftarrow resultado do cálculo $14 \% 3$

Então, é simples:

- [a] Se você escreve $14 / 3$, você obtém o resultado **4.666666666666667**. [float, claro!]
- [b] Se você escreve $14 // 3$, você obtém o resultado **4** [o quociente] em formato integer.
- [c] Se você escreve $14 \% 3$, você obtém o resultado **2** [o resto da divisão] em formato integer.

5.2. Prioridades na Frente!

Algumas vezes, somos obrigados a construir no Python expressões matemáticas com mais de um operador, como, por exemplo: `print(23 + 38 * 2 ** 5 // 8 % 4)`.

A pergunta que fica no ar é: como o Python entende [e resolve] uma expressão como essa? É só ler as operações da esquerda para a direita e calcular uma a uma em sequência?

Era o que eu já ia perguntar, João: como resolver essa complicação aí?

Calma, caro leitor, estimada leitora! O Python tem uma forma "organizada" de ler as expressões matemáticas e de resolvê-las... Acompanhe a tabela abaixo:

Primeiro...	Python resolverá todas as operações com o ** [potência].
Depois...	Python resolverá todas as operações de * , / , // e % - elas têm a mesma prioridade.
Por fim...	Python resolverá todas as operações de + e - . Elas têm a mesma prioridade entre si.

Então, vejamos o código a seguir.

```
30 + 160 // 2 ** 5 / 5 - 12
```

A primeira coisa que o Python vai resolver é a potência, ou seja, o **2 ** 5**. Isso dá **32**. [integer]

```
30 + 160 // 32 / 5 - 12
```

Depois, todas as multiplicações, divisões e restos em ordem, da esquerda para a direita. Então, fará logo a resolução do trecho **160 // 32**. O resultado disso é **5** [integer] – divisão inteira.

```
30 + 5 / 5 - 12
```

Agora, resolve-se o **5 / 5**, que resulta em **1.0**. [float] – lembre-se: divisão simples retorna um float.

```
30 + 1.0 - 12
```

Agora é só soma e subtração. O resultado disto é **19.0**. [float]

5.2.1. Alterando as Prioridades

Normalmente, o Python fará as operações matemática na ordem das prioridades que eu mostrei acima. Mas você pode fazer o Python realizar as operações em outra ordem: é só envolver com **()** [parênteses] a parte da equação que você deseja que o Python resolva primeiro. Vamos ver?

```
40 - 8 * 3
```

O Python vai resolver essa expressão assim: [a] primeiro, resolve a multiplicação **8 * 3**, que dá **24**; [b] depois, resolve a subtração **40 - 24**, que dá **16**. O resultado, portanto, é **16**!

Se colocarmos os parênteses envolvendo a operação **40 - 8**, o Python será obrigado a resolvê-la primeiro, antes mesmo da multiplicação, ou seja:

```
(40 - 8) * 3
```

A resolução desta expressão será: [a] **40 - 8** é resolvido primeiro, e dá **32**. [b] A multiplicação **32 * 3** é resolvida posteriormente, resultando em **96**. O resultado, portanto, é **96**.

Você poderá precisar, se seu cálculo for muito complexo, de vários níveis de parênteses. Um dentro do outro, como veremos no exemplo a seguir:

```
3 * (10 // (2 + 3) + 8 * (4 - 3) + 2 ** (8 // (3 + 1) + 1))
```

Como o Python vai resolver isso? Acompanha a sequência:

[a] Primeiro, resolvem-se todos os parênteses mais internos:

```
3 * (10 // (2 + 3) + 8 * (4 - 3) + 2 ** (8 // (3 + 1) + 1))  
3 * (10 // 5 + 8 * 1 + 2 ** (8 // 4 + 1))
```

[b] Agora, podemos analisar os cálculos que têm prioridades dentro de cada parêntese restante:

```
3 * (10 // 5 + 8 * 1 + 2 ** (8 // 4 + 1))  
3 * ( 2 + 8 + 2 ** ( 2 + 1))
```

[c] Agora, o Python irá resolver aquele parêntese interno e aquela soma entre os dois itens iniciais:

```
3 * (2 + 8 + 2 ** (2 + 1))  
3 * ( 10 + 2 ** 3 )
```

[d] Hora de resolver a potência, né? Depois somá-la ao 10 inicial para resolver os parênteses principais:

```
3 * (10 + 2 ** 3)  
3 * (10 + 8)  
3 * (18)  
54
```

Lembre-se de que é o Python que vai fazer esses cálculos. Não Você! É importante, porém, que você saiba exatamente como ele faz esses cálculos, porque você saberá escrevê-los de maneira a obter o resultado que deseja. Afinal, escrever uma equação de maneira errada trará resultados diferentes!

5.3. Vamos Exercitar a Lição 5

5.3.1. Qual o resultado dos seguintes cálculos? [escreva-os aqui]

- [a] $30 + 20$
- [b] $50 - 12$
- [c] $80 * 2$
- [d] $30 / 2$
- [e] $100 - 25 * 2$
- [f] $23 + 8 * 2 / 4$
- [g] $(90 - 10) * 2 / 5$
- [h] $7 // 2$
- [i] $11 \% 3$
- [j] $(4 + 17) \% 3$
- [k] $9 + 25 // 3$
- [l] $4 ** 3$
- [m] $80 + (33 // 4) ** 2$
- [n] $40 - (3 * 3) ** 2 // 4$
- [o] $80 / 2 ** (5 \% 2)$

5.3.2. O que vai ser apresentado na tela quando o programa abaixo for executado?

```
print("Calculadora " * 3)
print('Primeiros', 3 * 'testes ')
print(80 + 40 // 3 % 3)
print((100 - 60) / 20 + 3 ** 2)
print("Salva de", 7 * 3, "tiros")
print((3 * ((3 + 4 % 3) * 2 + 3) - 12) * 'bum ')
```

Escreva a saída [o resultado que aparecerá na tela do programa] aqui em baixo:

Curte lá meu Facebook! Tem muita novidade: www.facebook.com/ProfessorJoaoAntonio.

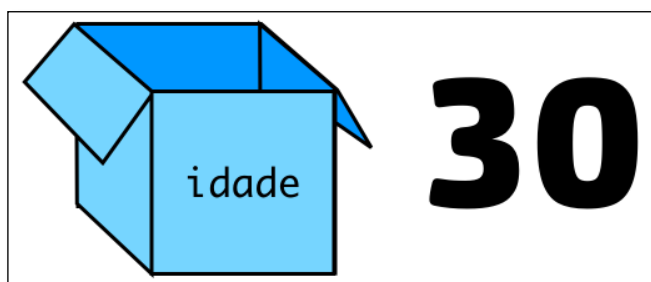
Lição 6. Variáveis

6.1. O que são Variáveis?

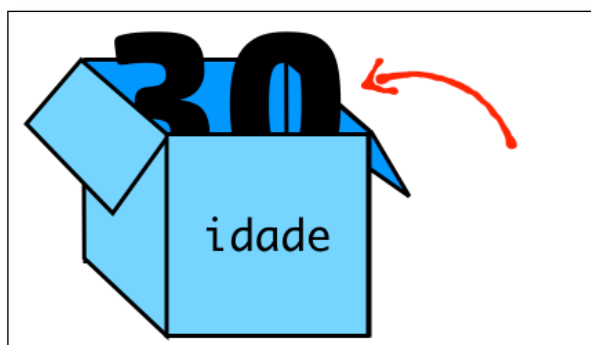
Variáveis são partes importantes de qualquer linguagem de programação. **Uma variável é um pequeno "espaço na memória do computador" utilizado para armazenar dados.**

Variáveis são utilizadas para que possamos usar os dados contidos nelas várias vezes.

Variáveis são como "caixas" onde podemos guardar dados para usá-los depois [como as caixas dos enfeites de Natal que todos os anos nós abrimos para usar os efeitos].



No exemplo ao lado, **o número 30 é um dado**, ou seja, um valor. E **idade é uma variável**: é um "local" onde podemos guardar esse valor para usar depois.



Agora, o dado 30 foi colocado dentro da variável idade. Sempre que "apontarmos" para a variável idade, o número 30 nos será devolvido como resposta.

6.1.1. Criando uma Variável

Para declarar [criar] uma variável, você só precisa dar a ela algum valor.

Declarar uma Variável: é dizer ao programa que ela existe – ou seja, é **criá-la**. No Python, quando você atribui um valor a uma variável pela primeira vez, ela é automaticamente criada.

Atribuir um Valor a uma Variável: é colocar algum valor dentro dela.

Lembre-se: para usar uma variável no seu programa [mencioná-la em cálculos, por exemplo], precisamos criá-la primeiro. Para criá-la, basta dar-lhe um valor pela primeira vez.

Mas como se faz isso no programa, João?

É bem fácil, amigo leitor, amiga leitora! Basta usar a expressão **nome_da_variável = valor**.

```
idade = 30
```

Esse é o código usado no Python para fazer o que a figura anterior mostra.

Para que você possa lembrar melhor, entenda o sinal de **=** como sendo a palavra "**recebe**". Então, o código

```
distancia = 300
```

pode ser lido como "**a variável distancia recebe o valor 300**".

João, posso dar qualquer nome para uma variável?

Não! Há algumas regras que devem ser respeitadas.

- [a] Nomes de variáveis devem conter apenas letras, números e `_` [underline].
- [b] Nomes de variáveis não podem começar com números

```
idade = 30
idade2 = 28
idade_cachorro = 8
_idade = 10
3idade = 18      # Isso dá erro [não funciona]
```

O Python é uma linguagem **case-sensitive** [significa que faz diferença entre maiúsculas e minúsculas]. Portanto, a variável **Idade** é diferente da variável **idade**, que é diferente da variável **IDADE**.

***Cuidado:** às vezes, o programador nomeia a variável com letra maiúscula e depois tenta usá-la com letra minúscula – o Python não vai aceitar isso! Preste atenção a como escreve os nomes das variáveis! Afinal, você não pode mencionar uma variável que não foi declarada!*

6.1.2. Usando Variáveis

Depois de criar uma variável, é possível usá-la em qualquer expressão apenas se referindo ao nome dela. **Usar uma variável, portanto, é mencioná-la.** Veja o código a seguir:

```
a = 30      # Aqui, a variável a está sendo declarada
b = 40      # Aqui, b variável a está sendo declarada
print(a + b)  # Aqui, as variáveis a e b estão sendo mencionadas
```

Perceba que eu criei a variável **a** com o valor 30, depois criei a variável **b** com o valor 40. Por fim, usei a função `print()` mencionando a soma entre **a** e **b**. O resultado do código acima será

```
70
```

Lembre-se de que não é possível usar [mencionar] uma variável que nunca foi declarada antes. Isso vai gerar um erro. Veja esse código:

```
a = 10
print(a + b)
```

O Python vai, com certeza, reclamar que não consegue entender o que é **b**. O código acima está mencionando uma **variável b** que nunca foi declarada antes, ou seja, para o programa, **b** não existe!

Lembra que para **criar [declarar]** uma variável, é necessário apenas atribuir um valor para ela.

Depois de atribuir um valor a uma variável, declarando-a, é possível modificar o conteúdo da variável quantas vezes você quiser. Basta, para isso, atribuir novamente outro valor a ela.

```
idade = 30
idade = 43
idade = 28
print(idade)
```

O resultado do código acima é

```
28
```

Perceba que, primeiramente, havíamos colocado o valor **30** na variável **idade**. Depois, substituímos ele pelo valor **43** [e o 30 foi esquecido]. Por fim, atribuímos o valor **28** à variável, fazendo com que ela esqueça o valor 43. Atualmente, portanto, a variável **idade** possui o valor **28**.

Ô João, então quer dizer que sempre que a gente atribui um novo valor a uma variável que já tinha algum valor, o valor anterior é substituído pelo novo?

Sim! Quando atribuímos um novo valor a uma variável, esse novo valor substitui completamente o valor que havia anteriormente. **Sai o velho, entra o NOVO!**

Mas você pode substituir o valor antigo por um valor que é atualização dele [é o que eu chamo de "levar em consideração" o valor anterior]. Veja um exemplo abaixo:

```
idade = 30
idade = idade + 1
print(idade)
```

O nosso programa funciona assim:

- [a] Primeiro, a variável **idade** recebe o valor 30;
- [b] Depois, somamos o valor contido em **idade** [que já é 30] com o número 1 e a variável **idade** recebe este novo valor. Ou seja, **idade** agora tem 31 [substituiu o 30].
- [c] Mandamos exibir o valor atual da variável **idade** com a função **print()**.

O resultado apresentado será

```
31
```

Podemos fazer isso com qualquer outra operação aritmética. Vejamos alguns exemplos:

```
a = 10
b = 20
c = 30
d = 40
```

```
a = a - 3
b = b * 2
c = c / 4
d = d % 6
print(a, b, c)
```

Atenção: o comando **print(a, b, c, d)** vai mostrar na tela, lado a lado, separados por espaços, os valores das variáveis **a**, **b**, **c** e **d**. O resultado deste programa será a apresentação, na tela, dos valores:

```
7      40      7.5    4
```

Podemos fazer esta operação de adicionar, subtrair, multiplicar, dividir ou calcular resto com o valor que já havia na variável usando uma outra forma de escrever: **+=**, **-=**, ***=**, **/=**, **//=**, **%=** e ****=**.

Escrever **num = num + 1** é o mesmo que escrever **num += 1**. Do mesmo modo que escrever a expressão **desconto = desconto - 10** é o mesmo que escrever **desconto -= 10**.

Então, o código:

```
a = 10
b = 20
c = 30
d = 40
a += 5 #É o mesmo que a = a + 5
b *= 3 #É o mesmo que b = b * 3
c /= 10 #É o mesmo que c = c / 10
d **= 2 #É o mesmo que c = c ** 2
print(a, b, c, d)
```

resultará em

```
15      60      3      1600
```

E só para lembrar, meu amigo leitor, minha amiga leitora, esses sinais [**+=**, **-=**, ***=**, **/=**, **//=**, **%=** e ****=**] devem ser escritos necessariamente juntos [**não se admitem espaços entre eles, ok?**].

6.1.3. Mais Detalhes sobre variáveis

Vamos analisar mais algumas linhas de atribuição de variáveis? Dá uma olhada aqui:

```
frase = "palavra"
palavra = "hoje"
hoje = frase
amanha = palavra + hoje
```

Na primeira linha, a **variável frase** recebe a string **"palavra"**. Na segunda linha, a **variável palavra** recebe a string **"hoje"**. Depois, a **variável hoje** recebe o mesmo valor que está na variável **frase** [ou seja,

a string "palavra"]. Por fim, a **variável amanhã** recebe a [soma] concatenação dos valores existentes nas variáveis palavra e hoje, ou seja, a string "hoje" concatenada com a string "palavra".

Ao término deste programa, se pedirmos:

```
print(frase, palavra, hoje, amanhã)
```

Vai trazer o resultado:

```
'palavra'      'hoje'      'palavra'      'hojepalavra'
```

Não confunda! Olha os dois exemplos abaixo:

```
var1 = "hoje"  
var2 = hoje
```

A **variável var1** recebe a **string "hoje"**. A **variável var2** recebe o conteúdo que está na **variável hoje**. Perceba, na linha da var2, a ausência de aspas, indicando que aquele **hoje** é o nome de uma variável e não uma string.

6.1.4. Atribuindo Valores a Múltiplas Variáveis

Uma das coisas que você poderá usar várias vezes no Python é que você pode atribuir valor a mais de uma variável numa única linha. Basta separar as variáveis por vírgulas e os valores também! Ou seja, o código:

```
nome, idade, altura = "João", 42, 1.90
```

é a mesma coisa que digitar

```
nome = "João"  
idade = 42  
altura = 1.90
```

Mais um exemplo para ilustrar essa técnica:

```
var1, var2 = 20, 30  
var1, var2 = var2+var1, var2*var1  
print(var1, var2)
```

Neste exemplo, inicialmente, a variável **var1** recebe **20** e a variável **var2** recebe **30**. Na segunda linha, a variável **var1** recebe **a soma entre var2 e var1**, ou seja, **30 + 20**, ou seja, **50** e a variável **var2** recebe **a multiplicação entre var2 e var1**, ou seja, **30 * 20**, ou seja, **600**.

O comando print(var1, var2) vai mostrar, portanto:

```
50      600
```

6.2. Chegou a Hora de Exercitar sobre a Lição 6

6.2.1. Quais os resultados dos códigos abaixo?

```
velho, novo = 17, 30
problema = novo - velho
problema = problema * 2
velho -= 7
print(novo, velho, problema)
```

```
eco = "Mises"
times = 3
times = times * 2
more = times + 4
print(eco * more)
```

```
base, expoente = 4, 10
expoente = base // 2
print(base ** expoente)
```

```
altura, base = 10, 20
altura, base = base * 3, altura + 5
area = base * altura
print(area, base*3, altura+10)
```

```
raio, cons = 15, 3
cons *= 2
raio += 2
resposta = cons + raio % 3
print(resposta, cons + raio, raio ** 2)
```

Lição 7. Alguns Detalhes a Mais

7.1. Comentários e Erros – Programa de Exemplo

Vamos fazer um programa simples que armazene dois números em duas variáveis diferentes e apresente vários cálculos aritméticos usando ambos.

```
a, b = 10, 2
print("=" * 30)
print("Calculadora Simples")
print("=" * 30)
print("A soma entre eles é: ", a + b)
print("A subtração entre eles é:", a - b)
print("A multiplicação entre eles é:", a * b)
print("A divisão entre eles é:", a / b)
print("A divisão inteira entre eles é:", a // b)
print("O resto da divisão entre eles é:", a % b)
print(a, "elevado à", b + "a", "potência é", a ** b)
```

O resultado deste programa simples é:

```
=====
Calculadora Simples
=====
A soma entre eles é:  12
A subtração entre eles é: 8
A multiplicação entre eles é: 20
A divisão entre eles é: 5.0
A divisão inteira entre eles é: 5
O resto da divisão entre eles é: 0
10 elevado à 2a potência é 100
```

Mudando os valores das variáveis a e b, é claro, você mudará os resultados do programa.

7.1.1. Comentários Curtos

Vamos agora aprender a "comentar" nosso código. Escrevemos comentários para entendê-lo melhor em caso de termos de lê-lo posteriormente – sempre é bom criar um "manual de instruções" no código.

Para fazer **comentários curtos** [uma linha apenas], use o caractere # [hashtag, ou "jogo da velha"].

Tudo aquilo que você escrever após este símbolo é automaticamente entendido como comentário.

Atenção: comentários são trechos de texto que o Python **não vai obedecer**. Esses textos existem no código em si, mas **nunca serão executados**. São apenas para "orientar" o programador.

Veja um programa simples com comentários:

```
# Aqui declaramos as variáveis que vamos usar
a = 20
b = 30
# Aqui um pequeno "cabeçalho" para o programa
print("=" * 30) # Repita o sinal de "=" 30 vezes
print("Calculadora")
# Aqui começam os cálculos
a = b + c # Aqui vai dar um erro!
b = a + 30
```

Como você pôde perceber, comentários podem ser escritos numa linha única [sozinhos] ou em uma linha que possui comandos [tudo o que aparecer depois do # será entendido como comentário].

Agora, olhe o código novamente e perceba que na penúltima linha há um comentário que diz "**Aqui vai dar um erro!**" – você pode imaginar por quê?

Deixa eu ver, João! Será que é porque a variável c está sendo requisitada, mas ela nunca foi declarada antes?

Exatamente!!! Muito bem! Não é possível pedir que a variável **a** receba a soma de **b** com **c**, se nós simplesmente **nunca ouvimos falar de c**. A variável **c** simplesmente não existe porque ela nunca foi declarada! [é um erro bastante comum entre os programadores – até os mais experientes].

7.1.2. Comentários Longos [Várias Linhas]

Para fazer comentários longos, textos de documentação grandes, você pode escrever todo o comentário desejado entre `"""` [três aspas duplas] ou `'''` [três aspas simples]. Coloque-os **antes** e **depois** do texto a ser digitado e todo ele será desconsiderado – totalmente ignorado [será considerado comentário].

Veja um exemplo de um programa simples:

```
"""
Este programa foi criado para exemplificar o uso dos comentários
de várias linhas. Basta colocar o trecho que será usado como comentário
entre grupos de 3 aspas. Tanto faz se são aspas duplas ou aspas simples.
"""
a = 90
b = 20
c = 30
print(a + b - d) # Vai dar erro porque não declaramos a variável d
# Não podemos usar uma variável que não tenha sido declarada antes
```


As aspas triplas também podem ser usadas para designar uma string [texto] com mais de uma linha, por exemplo, para a função print(). Veja um exemplo:

```
print("""Pequeno Poema
Subi num pé de manga
Pra tirar abacaxi.
Como não era tempo de morango,
Roubaram minha bicicleta.""")
```

7.2. *Vamos Exercitar sobre a Lição 7*

7.2.1. Quais os resultados dos programas abaixo? Eles dão erro? Eles funcionam?

```
# Programa para repetir um valor textual várias vezes
Letra = '''A "Casa" Amarela. '''
numero = 3
print(Letra * numero)
```

```
# Programa de Exemplo para multiplicar variáveis
1num, 2num = 10, 20
print(1num * 2num)
```

```
# Programa de Exemplo para multiplicar variáveis
    Agora com nomes corretos das variáveis
x, y = 10, 20
print(x * y)
```

Lição 8. Valores Booleanos [Falso / Verdadeiro]

8.1. O que é Booleano?

Já conhecemos alguns tipos de dados que o Python entende, como integer, float e string. Chegou a hora de conhecer um outro tipo de dado: o tipo **boolean** [ou "booleano"]. Dados em formato booleano só podem receber dois valores contrários: **False** [falso] ou **True** [verdadeiro].

Note bem, amigo leitor, o valor **False** é diferente de **"False"**. False é um conceito, uma ideia, um estado de falsidade. O valor **"False"** é um texto, uma string [se está entre aspas, é uma string].

Da mesma forma que **True** é uma ideia que indica um estado de verdadeiro e **"True"** é uma string.

Podemos informar que uma variável é diretamente True ou False do mesmo jeito como fazemos em qualquer outro tipo de valor:

```
teste = False
aprovado = True
```

Atenção: ao se referir aos valores booleanos, você deve escrevê-los **True** e **False** [com a primeira letra maiúscula]. Escrevê-los de outra forma faz o Python não entender como booleano.

Outra forma de obter os valores False e True é usar proposições [normalmente **expressões que usam os operadores de comparação**].

Proposição: é uma expressão que só pode assumir dois valores: ou ela é verdadeira, ou é falsa.

Operadores de Comparação: são sinais que realizam a comparação entre valores e/ou expressões.

Vamos a um exemplo?

A expressão **20 > 30** quer dizer **"vinte é maior que trinta"**. Nós sabemos que isso é **False**. Do mesmo modo que **20 < 30** significa **"vinte é menor que trinta"**. Isso é **True**.

Então, olha que lindo! Operadores de comparação são justamente esses sinais [**>** e **<**, nos exemplos acima]. Com esses operadores, nós comparamos o que está antes do operador com o que está depois.

Vamos conhecer todos eles?

20 > 30 ["vinte é maior que trinta"]

20 < 30 ["vinte é menor que 30"]

20 >= 30 ["vinte é maior ou igual a 30"]

20 <= 30 ["vinte é menor ou igual a 30"]

20 == 30 ["vinte é igual a 30"]

20 != 30 ["vinte é diferente de 30"]

Atenção: o sinal de comparação "igual" é necessariamente um **== [igual duplo]** porque o uso do sinal de **= único** é para atribuir valores a variáveis, como já estamos fazendo há algum tempo.

Então, podemos testar algumas proposições?

```
a = 20
b = 10
c = 30
print(a > b)
print(c <= b)
print(a + 2 * c) # Aqui não é uma proposição porque não tem comparação
print(a * 2 > c + b)
```

O resultado deste programa é que aparecerá na tela o seguinte:

```
True
False
80
False
```

Portanto, o valor **False** indica que a proposição é mentira. O valor **True** indica que a proposição é verdade. Os valores "True" e "False" são apenas textos [strings].

Podemos usar valores booleanos em uma série de momentos no Python, mas vamos usá-las especialmente em **estruturas de decisão** [condicionais] e **estruturas de repetição** [loop].

8.2. Comparação com Textos

Estamos acostumados a entender as operações de comparações com números, pois é fácil entender, por exemplo, a expressão **8 > 12** e imediatamente compreender que ela é **False**.

Mas podemos fazer comparações entre valores do tipo string também – e o Python é um tanto "esperto" em relação a elas. Inicialmente, vamos comprar textos usando "igual" e "diferente":

```
"Casa" == "Escola"
"Casa" != "Escola"
```

O resultado dos dois comandos acima será:

```
False
True
```

Isso porque a String "Casa" é diferente da string "Escola" [deu pra notar, né?].

Mas uma coisa que poucos sabem [e parece não ter lógica] é que podemos usar comparadores como "maior que" e "menor que" também com as strings.

Peraí, João, aí não dá para aceitar! Como é que o Python vai decidir se "Casa" é maior que "Escola"?

Pela **Posição no Dicionário** das strings, amigo leitor, amiga leitora! Ou seja, os comandos:

```
"Casa" < "Escola" # "Casa" vem antes de "Escola" no dicionário
"Pernambuco" < "Piauí" # "Pernambuco" vem antes de "Piauí" no dicionário
"Mises" >= "Marx" # "Mises" vem depois ou igual a "Marx" no dicionário
"Liberdade" <= "estado"
```

Resultarão, respectivamente, em:

```
True
True
True
False
```

8.3. Vamos Exercitar a Lição 8

8.3.1. Qual o valor da variável teste nos vários casos a seguir?

teste = 80 + 40

teste = 80 < 40

teste = 2 * 40 > 4 * 10

teste = 4 * 10 - 30

teste = 4 * 10 - 30 < 20 * 2

teste = 80 == 80

teste = "Indivíduo" > "Estado"

teste = "Imposto" >= "Liberdade"

teste = 30 * 12

teste = 100 * 2 > 10 ** 3

teste = 20 + 5 != 5 ** 2

teste = 80 % 9 == 2 ** 3

teste = 50 + 10 <= 20 * 3

teste = 40 + 30

teste = 2 * 4 ** 2 + 10

teste = 2 ** 5 > 3 * 10

teste = 80 + 15 < 10 ** 2 - 5

Lição 9. Funções Úteis

9.1. O Que são Funções?

Funções são pequenos trechos de código que podem ser reutilizados. Numa lição posterior, aprenderemos a criar as nossas próprias funções.

Praticamente, todas as funções são escritas do mesmo jeito: **nome_da_função(argumentos)**. Você precisa conhecer o nome da função para usá-la – e você precisa saber quais argumentos ela aceita para ela funcionar corretamente.

Pense numa função **lavar()** que serve para, digamos, lavar objetos. Aí, usar **lavar(carro)** poderia ser, hipoteticamente, a forma de mandar lavar o carro e **lavar(bicicleta)** serviria para mandar lavar a bicicleta. [tá, ok, isso é só um exemplo "bem besta" para explicar funções].

A função **print()** já conhecemos – ela é usada para imprimir [na tela] aquilo que estiver colocado como argumento [dentro dos parênteses]. Vamos conhecer mais algumas, começando pelas...

9.2. Funções de Conversão de Dados

A função **int()** serve para converter um número float ou string para um número no formato integer.

Ou seja, o comando **int(7.5)** vai resultar em **7**. O comando **int("8")** vai resultar em **8**. Note que o primeiro valor era um número float e o segundo exemplo era um string.

A função **float()** converte um número integer ou uma string em um número no formato float.

Exemplos? Agora mesmo! Se você escrever **float(8)**, isso resulta em **8.0**. Se você escrever **float("10")**, isso vai gerar, como resultado, o número **10.0**.

Atenção: nem a função **int()**, nem a função **float()** vão funcionar se o valor string não for "escrito só com números". Por exemplo, vai aparecer erro se você tentar usar **int("casa")** ou **float("escola")** [não tem como converter esses aqui pra números, né?]

A função **str()** é usada para converter qualquer valor em uma string [texto]. Pode ser usada com números integer ou float.

Então, usar a função **str(18)** vai resultar em **"18"**, e usar a função **str(10.0)** vai resultar em **"10.0"**.

A função **bool()** é usada para converter qualquer valor [número ou string] para um valor booleano [**True** ou **False**]. A norma, porém, é um tanto "curiosa":

[a] Se tentar converter o **número 0** ou uma **string vazia**, o resultado será **False**.

[b] Em qualquer outro caso, o resultado será **True**.

Portanto, nos casos a seguir:

```
bool("casa")
bool(34)
bool(0)
bool("")
```

teremos os resultados seguintes

```
True
True
False
False
```

9.3. Funções Numéricas Gerais

A função **sum()** é usada para somar dois ou mais números. Ou seja, se você usar **sum([10,20])**, o resultado será **30**. O argumento da função **sum()** tem que ser escrito **entre colchetes**, o que, para o Python, significa uma **lista**, o que chamamos de objeto **iterável** [vamos ver isso adiante].

O resultado de **sum([1, 2, 3, 4, 5, 6])** será **21**. Nem preciso dizer que o mesmo que escrever **1 + 2 + 3 + 4 + 5 + 6**, né?

A função **max()** pode ser usada com vários valores [sem colchetes] ou com uma lista, como a função **sum()**. A função **max()** retorna [resulta] **o maior valor** encontrado nos argumentos.

Então, a função **max(2, 40, 5, 82, 13)** retornará o valor **82**.

***Um aviso:** a expressão "tal função retornará tal valor" significa que "tal função resultará em tal valor". Ou seja, o verbo "retornar" é usado como sinônimo de "resultar". Nós vamos aprender que nem toda função retorna um valor... mas quando esse verbo for usado, é porque a função irá trazer resultados.*

A função **min()** é irmã da **max()**. Ela retorna **o menor valor** apresentado nos argumentos. Portanto, a função **min(2, 40, 5, 82, 13)** retornará o valor **2**.

A função **round()** serve para **arredondar um número** para uma determinada quantidade de casas decimais. Para isso, escreva **round(número, núm_casas)**. Um exemplo? Claro! Aqui vai: O comando **round(12.3945, 1)** vai retornar **12.4**, que é o número 12.3945 sendo mostrado com apenas 1 casa decimal.

Quando você não informa o número de casas, o Python corta todas [ou seja, converte o número para um integer].

Mais uma função? Que tal a função **len()**? A função **len()** conta a quantidade de **itens de uma lista** ou de **caracteres em uma string**.

Ou seja, a função `len("Mises")` vai retornar **5**, porque "Mises" tem 5 caracteres.

Escrever a função que retorna valor sozinha numa linha do editor de código não adianta: ou você aponta o resultado da função para uma variável, ou manda imprimir na tela por meio da função `print()`, como mostramos aqui:

```
num_maximo = max(1, 3, 9, 5, 2, 8) # Atribuindo a uma variável  
print(int(20/6)) # Imprimindo direto na tela
```

9.4. Vamos Exercitar sobre a Lição 9

9.4.1. Quais os resultados das Funções abaixo?

```
int(3.45)
```

```
int("5")
```

```
max(2, 5, 6, 1, 4, 3)
```

```
str(123)
```

```
min(20, 3 + 4, 15, 5 * 2)
```

```
bool("Escola")
```

```
int(20 / 3)
```

```
float(12)
```

```
float(int(3.5) + int(4.5))
```

```
sum([3, 4, 7*2, 5])
```

Mais Cursos?: *Visita meu site:* www.ProfessorJoaoAntonio.com

Lição 10. Recebendo Dados do Usuário

10.1. A Função de Entrada de Dados [Input()]

Usamos a função **print()** para enviar mensagens ao usuário [ou seja, mostrar coisas na tela para que o usuário leia].

Algumas vezes, porém, precisaremos pedir dados ao usuário [esperar que ele digite algo]. Isso é trabalho para a função **input()**.

A função **input()** é usada para a entrada dos dados [assim como a **print()** é para a saída].

Atenção: chamamos de **"entrada de dados"** o ato de o usuário inserir dados no computador.

Sempre que você precisar que o usuário informe algo, como o nome dele ou a idade, você deverá usar a função **input()**.

A função **input()** é usada sempre deste jeito: **variável = input(mensagem)**. Ou seja, a função **input()** deve ser usada numa **atribuição de variável**, para que o valor que o usuário digitar seja armazenado diretamente na variável apontada. O argumento **mensagem** é o texto que aparecerá para o usuário antes de ele inserir o dado pedido.

Vamos a um exemplo:

```
# Programa para exibir Nome e Idade
nome = input("Digite o seu nome:") # Pede o nome do usuário
idade = input("Digite a sua idade:") # Pede a idade dele

# Exibir uma linha formada por 30 asteriscos
print("*" * 30)

# Exibir os Dados
print(nome + " tem " + idade + " anos de idade.")
```

O resultado deste programa é:

```
Digite o seu nome: João Antonio
Digite a sua idade: 42
*****
João Antonio tem 42 anos de idade.
```

Depois de inserir cada valor pedido, o usuário deverá pressionar **ENTER** para confirmar. Portanto, considere que depois de ter digitado **João Antonio**, o usuário pressionou **ENTER**. Considere também que depois de ter digitado **42**, o usuário também pressionou **ENTER**.

Um alerta importante: todas as vezes que você usa a função `input()`, o dado que ela captura é, necessariamente, uma string [mesmo que você digite um número como a idade de alguém].

Isso significa que se você quiser usar um dado capturado pela função `input()` para fazer cálculos, é bom passar ele numa função de conversão antes. Por exemplo:

```
idade = int(input("Digite a idade:"))
```

No exemplo acima, o resultado da função `input` [inserido pelo usuário] será usado como parâmetro da função `int()`. O que será armazenado na variável `idade` é um número integer.

Esse código deixa isso mais claro:

```
>>> idade = input("Digite a idade:")
Digite a idade: 42 # O usuário digitou aqui!
>>> print(idade)
'42' # Note que aqui o 42 é um texto [está entre aspas simples]
>>> idade = int(idade) # Aqui convertemos a variável idade
>>> print(idade)
42 # Note que aqui o 42 é número pois não tem aspas
```

10.2. Vamos Exercitar acerca da Lição10

10.2.1. Escreva um programa que peça o nome e a idade do usuário e, em seguida, imprima na tela a mensagem "<nome> tem <idade> anos de idade" [substitua <nome> e <idade> pelos dados que o usuário inserir].

10.2.2. Escreva um programa que receba dois números, a largura e a altura de um retângulo e apresente na tela a mensagem "A área do retângulo é <Área> ." [Substitua <Área> por $\text{largura} * \text{altura}$].

10.2.3. Qual o resultado, impresso na tela, dos programas abaixo?

```
num1 = input('Digite o primeiro número:') # Usuário digita 10
num2 = input('Digite o segundo número:') # Usuário digita 20
print(num1 + num2)
```

```
num1 = input('Digite o primeiro número:') # Usuário digita 40
num2 = int(input('Digite o segundo número:')) # Usuário digita 5
print(num1 * num2)
```

```
print("=" * 10)
print("Calculadora")
print("=" * 10)
num1 = int(input('Digite o primeiro número:')) # Usuário digita 30
num2 = int(input('Digite o segundo número:')) # Usuário digita 7
print(int(num1/num2))
```

Lição 11. Listas

11.1. O que são listas?

O Python nos permite usar um tipo de "variável" interessante: **as listas**. Listas são variáveis que permitem armazenar vários valores. Ou seja, uma lista é uma **estrutura de dados**, como se fossem variáveis com "muitas gavetas".

Veja os exemplos a seguir:

```
nome = "Ludwig von Mises"
idade = 103
```

Note que a variável **nome** só consegue guardar uma string e a variável **idade** só consegue guardar um número [ou seja, normalmente, há apenas um dado em cada variável].

Para criar uma lista, você atribui os valores que quiser inserir na lista dentro de um par de **colchetes**, ou seja, esses símbolos aqui: **[]**. Os elementos dentro da lista são **separados por vírgulas**.

Vamos criar uma lista chamada **economistas** no exemplo a seguir:

```
economistas = ["Hayek", "Mises", "Menger", "Rothbard"]
```

Se você pedir para mostrar a lista **economistas**, vai aparecer isso:

```
>>> print(economistas)
['Hayek', 'Mises', 'Menger', 'Rothbard']
```

Você pode pensar numa lista como sendo uma "tabela" com uma linha de dados apenas. Olha o exemplo a seguir para ilustrar a lista do exemplo acima:

economistas			
Hayek	Mises	Menger	Rothbard

11.1.1. Adicionando mais um item à Lista

Para adicionar mais um valor à lista, usa-se o método **.append()**.

Método: é uma função especial que é escrita como um "sobrenome" de um objeto. Um método é normalmente específico para um tipo de objeto. Usam-se métodos assim: **objeto.método()**.

O método **append(dado)** é usado apenas em listas. No caso do exemplo acima, usaríamos **economistas.append('Marx')** para adicionar a string 'Marx' no final da lista **economistas**. O resultado, ao pedir para exibir a lista **economistas** será:

```
>>> print(economistas)
['Hayek', 'Mises', 'Menger', 'Rothbard', 'Marx']
```

11.1.2. Apagando um Item da Lista

Para **apagar o último item** de uma lista [ou seja, tirar o Marx no nosso exemplo, já que ele não entendia bulhufas de economia], basta usar o método **.pop()**. Basta fazer o seguinte:

```
>>> economistas.pop()
'Marx' # O Python retorna o valor e o apaga da lista
>>> print(economistas)
['Hayek', 'Mises', 'Menger', 'Rothbard']
```

Note que, ao usar o método **.pop()**, o Python retorna (responde) com o valor que está sendo retirado da lista. Depois, ao mandar exibir a lista, dá para ver que o último item foi retirado – graças a Deus.

Ô João, e se eu quiser buscar [para exibir] apenas um item da lista, como posso fazer?

Você tem que usar a expressão **lista[índice]** para acessar diretamente um item da lista. Para explicar melhor, vamos entender que cada posição na lista tem um endereço próprio, que começa com o **número 0 [zero]**. Veja o exemplo para a lista abaixo:

economistas			
0	1	2	3
Hayek	Mises	Menger	Rothbard

Esta lista tem 4 posições [ou itens]. A **posição 0** contém a string **'Hayek'**, a **posição 1** contém a string **'Mises'** e assim por diante.

Quer mostrar na tela o conteúdo da posição 2 da lista? Olha abaixo:

```
>>> print(economistas[2])
Menger
```

Se você quiser alterar o conteúdo de uma posição, é só apontar para ele como se fosse uma variável e atribuir um novo valor:

```
>>> economistas[2] = 'Keynes' # Atribuimos o valor à posição 2
>>> print(economistas)
['Hayek', 'Mises', 'Keynes', 'Rothbard']
```

11.1.3. Apagando Qualquer Item da Lista [não só o último]

O método **.pop()** pode ser usado para apagar qualquer item da lista, não somente o último. Para isso, basta você indicar, dentro dos parênteses, qual o índice do item que você deseja apagar da lista. Veja o exemplo do código:

```
>>> print(economistas)
['Hayek', 'Mises', 'Menger', 'Rothbard']
>>> economistas.pop(1) # Retornará e apagará o item na posição 1
```

```
'Mises'
>>> print(economistas)
['Hayek', 'Menger', 'Rothbard']
```

11.1.4. Outros Métodos Interessantes para Listas

Vamos saber como inserir um item em uma posição qualquer na lista. Para isso, usamos o método **.insert(índice, valor)**. Esse método insere o valor desejado diretamente na posição indicada e "empurra" os demais valores para as posições posteriores.

```
>>> alunos = ['Joao', 'Pedro', 'Davi', 'Mateus', 'Miguel']
>>> print(alunos)
['Joao', 'Pedro', 'Davi', 'Mateus', 'Miguel']
>>> alunos.insert(3, 'Ana') # Insere 'Ana' na posição 3
>>> print(alunos) # insert() empurra os demais para posições posteriores
['Joao', 'Pedro', 'Davi', 'Ana', 'Mateus', 'Miguel']
```

Para apagar um item específico da lista, informando o valor em si, e não o índice, você pode usar o método **.remove(item)**. Note a diferença: no **.pop(índice)** você informa o índice da posição. No método **.remove(item)**, você informa o valor em si, não a posição.

```
>>> alunos.remove('Davi') # Remove 'Davi' da Lista
>>> print(alunos) # remove() puxa os demais para posições anteriores
['Joao', 'Pedro', 'Ana', 'Mateus', 'Miguel']
```

Um outro método importante é o **.reverse()**. Esse método inverte a ordem dos itens na lista.

```
>>> print(alunos)
['Joao', 'Pedro', 'Davi', 'Mateus', 'Miguel']
>>> alunos.reverse() # Inverte os itens da Lista
>>> print(alunos)
['Miguel', 'Mateus', 'Davi', 'Pedro', 'Joao']
```

Ainda há outro método a ser ensinado: o método **.sort()**. Este método organiza a lista em ordem [alfabética ou numérica]. O jeito mais simples é simplesmente usá-la sem argumentos, como a seguir:

```
>>> print(alunos)
['Joao', 'Pedro', 'Davi', 'Mateus', 'Miguel']
>>> alunos.sort() # Ordena [alfabética] os itens da Lista
>>> print(alunos)
['Davi', 'Joao', 'Mateus', 'Miguel', 'Pedro']
```

Apenas um lembrete: uma lista pode conter dados com tipos diferentes [strings, integers e floats, por exemplo]. **Uma lista pode até conter outras listas** [veremos depois].

Para finalizar, vamos conhecer o método **.count()**. Ele serve para contar quantas vezes um determinado valor aparece na lista. Veja o exemplo a seguir:

```
>>> lista = ['Mises', 'Hayek', 'Mises', 'Mises', 'Iorio', 'Iorio']
>>> lista.count('Mises')
3
```

Perceba que para usar o método `.count()`, você deve indicar o valor a ser contado dentro da lista. No exemplo acima, pedi para contar quantas vezes **'Mises'** aparece na lista **lista**.

11.2. O comando `del`

Uma outra forma de apagar um valor em uma lista é o comando **`del`**. Ele não é uma função ou método, mas um comando simples que deve ser usado diretamente no código, apontando para qual item da lista deseja excluir, veja o exemplo a seguir, que apaga o item de **índice 2** da **lista a**:

```
>>> a = [1, 2, 3, 4, 5]
>>> del a[2]      # Apagamos o item índice 2 da lista a
>>> print(a)
[1, 2, 4, 5]
```

Note como se escreve o comando **`del`**: **`del objeto_a_ser_apagado`**.

Outra diferença é que **o comando `del` não retorna o valor** do item apagado [como faz o **método `.pop()`**], ou seja, esse comando simplesmente apaga o item.

O comando `del` não pode ser usado apenas para listas, não – ele pode ser usado com variáveis simples, também. Se você aciona o comando `del` apontando para a variável em si [não somente uma posição dela], a variável inteira deixará de existir e o Python não a reconhecerá mais:

```
>>> a = "Menger"
>>> del a      # Apagamos a variável a inteira
>>> print(a)
Traceback (most recent call last):
  File "<pyshell#77>", line 1, in <module>
    a
NameError: name 'a' is not defined
```

Note que não foi possível imprimir na tela a **variável a** porque ela simplesmente **não existe mais**.

11.3. O Teste de Pertencimento `in`

Um recurso muito interessante para listas e outros tipos de dados múltiplos é o teste de pertencimento: uma forma de testar se um determinado valor está dentro daquela lista ou não. Usamos o operador **`in`** para fazer esse teste. O nome **`in`** é a mesma coisa que **"em"** em inglês. Portanto, ela serve para estar se um determinado valor existe **em** uma determinada lista. Olha como funciona:

```
>>> lista1 = [1, 2, 3, 4, 5]
```

```
>>> print(4 in lista1)
True
>>> print(8 in lista1)
False
```

Note que ao mandar imprimir **4 in lista1**, o resultado foi **True**: porque existe o valor 4 em lista1. Quando pedimos para imprimir o resultado de **8 in lista1**, o resultado foi **False**, porque não há 8 em lista1.

O resultado de uma expressão com **in** sempre é booleano!

11.4. Vamos Exercitar a Lição 11

11.4.1. O que será exibido na tela ao executar o código abaixo?

```
lista = [1, 5, 3, 4, 0, 2]
lista.pop(3)
lista.reverse()
a = lista[2] + lista[3]
lista.remove(5)
print(a + lista[3])
```

Aparecerá na tela:

11.4.2. Qual o conteúdo do objeto lista ao término do código a seguir?

```
lista = ['a', 'b', 'c', 'd']
lista.reverse()
lista.append('e')
lista.pop(2)
lista[0] = lista[3]
lista.remove('c')
```

lista = []

11.4.3. Qual o conteúdo do objeto lista ao término do código a seguir?

```
lista = [1, 5, 9, 13, 17, 21, 25]
del lista[3]
del lista[1]
lista[1] = lista[3]
lista.remove(25)
```

lista = []

Lição 12. Vamos Exercitar Muito?

12.1. Programa para Preencher uma Lista com 5 Nomes

Estamos organizando uma lista contendo os nomes dos 5 participantes que a escola levará para as Olimpíadas de Matemática. Para isso precisamos de um programa que peça ao usuário o nome de cada aluno escolhido e o coloque em uma lista. Depois de inseridos todos os nomes, o programa irá imprimir a lista completa. **Vamos construir o programa?**

```
alunos = [] # Isso cria uma lista vazia
nome = input("Digite o nome do aluno:") # Pegar o 1º Nome
alunos.append(nome)
nome = input("Digite o nome do aluno:") # Pegar o 2º Nome
alunos.append(nome)
nome = input("Digite o nome do aluno:") # Pegar o 3º Nome
alunos.append(nome)
nome = input("Digite o nome do aluno:") # Pegar o 4º Nome
alunos.append(nome)
nome = input("Digite o nome do aluno:") # Pegar o 5º Nome
alunos.append(nome)
print(alunos) # Imprime a lista inteira na Tela
```

Explicando o programa acima: o primeiro comando cria uma **lista vazia** [ou seja, sem itens] com o nome **alunos**. Para fazer isso, atribua colchetes vazios ao nome da lista a ser criada.

No comando seguinte [na segunda linha], acontece um **input** [que pede dados ao usuário] e coloca esse dado dentro da variável **nome**.

Na terceira linha, o conteúdo da variável **nome** é anexado na lista **alunos** [colocado no fim da lista, com o método **.append()**].

Nas linhas seguintes, repetem-se os atos de [a] pedir dado ao usuário; [b] guardar esse dado na variável **nome** e, em seguida, [c] anexar [adicionar] o conteúdo da variável **nome** no fim da lista **alunos**.

Ao final deste programa, a lista é impressa na tela. Simples, né?

O resultado do programa na tela será o seguinte:

```
Digite o nome do aluno: Joao
Digite o nome do aluno: Ana
Digite o nome do aluno: Pedro
Digite o nome do aluno: Mateus
Digite o nome do aluno: Miguel
```



```
['Joao', 'Ana', 'Pedro', 'Mateus', 'Miguel']
```

Agora é com você, ok?

12.2. Programa para Juntar e Calcular 5 Números

Precisamos criar um programa para receber 5 números inteiros, calcule a soma e a multiplicação entre todos eles e apresente, na tela: [a] a lista inteira, [b] a multiplicação entre eles e [c] a soma entre eles.

Dica: *comece pensando quantas variáveis você vai precisar usar...*

[Escreva o código aqui abaixo]

Tem mais aulas no meu Site: www.ProfessorJoaoAntonio.com

12.3. Programa para Juntar 6 Nomes e 6 Idades:

Que tal usar agora duas listas diferentes? Faça um programa que [a] peça um nome de seis pessoas; [b] peça a idade das 6 pessoas; [c] armazene isso em duas listas diferentes [uma para nome e outra para idade].

Ao final do programa, imprima as 6 frases "<nome> tem <idade> anos."

[Escreva o programa aqui embaixo] – teste o código e/ou acompanhe o vídeo de correção.

Telegram: Bora trocar ideias no grupo do Telegram? Meu username é: [@ProfessorJoaoAntonio](#).

Lição 13. Mais Sobre Listas

13.1. Lista e Muitos Tipos

Uma lista pode ser preenchida com dados de **diversos tipos diferentes**, como strings, integers e floats.

Uma lista não exige que todos os seus dados sejam do mesmo tipo! Veja isso:

```
>>> dados = [3, 'Casa', 120, 2.34, 'Texto', 99, 8, 3.9]
>>> print(dados[4])
Texto
>>> print(dados[0]*dados[4])
TextoTextoTexto
```

13.2. Lista de Listas

Já que uma lista recebe vários tipos de dados, é fácil aceitar que uma lista poderá receber, inclusive, **outras listas** como dados. Ou seja, é possível criar uma lista com várias listas dentro dela.

É normal chamar isso de **"listas aninhadas"** [**nested lists**, no inglês]. Isso significa "uma lista funcionando como 'ninho' das outras". Veja o exemplo a seguir:

```
>>> lista_principal = [['a','b','c'],[1,2,3],['xx','yy','zz']]
>>> lista_principal.append([9,10,11])
>>> print(lista_principal)
[['a','b','c'],[1,2,3],['xx','yy','zz'],[9,10,11]]
```

Note que a **lista_principal** é uma lista composta por 4 listas. Ou seja, cada item da lista_principal é uma lista diferente. Vamos "pintar" cada item para deixar mais fácil de entender:

```
[[['a','b','c'],[1,2,3],['xx','yy','zz'],[9,10,11]]
```

Isso pode parecer um tanto "inútil" ou "estranho" agora, amigo leitor, amiga leitora... mas em breve veremos muita utilidade para este recurso. Uma lista aninhada parece com isso aqui:

lista_principal			
0	a	b	c
	0	1	2
1	1	2	3
	0	1	2
2	xx	yy	Zz
	0	1	2
3	9	10	11
	0	1	2

A gente sabe que, para acessar um item de uma lista, a gente escreve **nome_da_lista[índice]**. Acontece que apenas a lista principal [a lista-mãe] tem nome... as demais listas [as filhas] são itens da principal. Então, como fazer para acessar, digamos, o item 'yy' do exemplo acima?

Bom, o item 'yy' está na **posição 1** da lista que está na **posição 2** da lista **lista_principal**. A forma para acessar aquela exata posição é **lista_principal[2][1]**, que significa "**a posição 1 da posição 2 da lista_principal**".

13.3. A função `list()`

Já aprendemos como criar uma lista atribuindo a uma variável diretamente os valores [entre colchetes e separados por vírgulas].

Outra forma de criar uma lista é usar a função **`list()`**.

Quando a função `list()` é usada assim, sem argumentos, ela cria uma lista vazia. Ou seja, escrever...

```
lista = list()
```

... é o mesmo que escrever...

```
lista = []
```

Podemos usar a função **`list()`** também colocando strings no argumento. Isso fará com que a lista seja criada tendo cada caractere da string como item. O exemplo a seguir mostra isso bem:

```
>>> lista = list('Mises')
>>> print(lista)
['M', 'i', 's', 'e', 's']
```

A lista pode ser criada, também, a partir de um objeto do tipo **sequência** [**iterable**, no inglês]. Um objeto sequência é um objeto formado por uma sequência numérica. A forma mais fácil de construir é por meio da função **`range()`**. Veja o exemplo a seguir [explico logo abaixo]:

```
>>> seq = range(10) # Cria uma sequência de 0 a 10
>>> print(seq)
range(0, 10) # Este é um objeto sequência – de 0 [zero] a 10 [dez]
>>> seq2 = list(seq) # Transforma a sequência numa lista.
>>> print(seq2)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] # Lista criada
```

Eu sei que você está estranhando que a lista só foi até o número 9. Não se preocupe: vou explicar direitinho na próxima lição.

13.4. Esvaziar uma Lista

Um outro método interessante para listas é o método `.clear()`. Este método apaga todos os itens de uma lista, deixando-a completamente vazia.

```
>>> lista = [1,2,3,4,5]
>>> lista.clear() # Neste momento, a lista foi esvaziada
>>> print(lista)
[]
```

13.5. Vamos Exercitar sobre a Lição 13

Observe o Código a seguir e responda às perguntas que se seguem a ele:

```
construtor = range(15)
lista = list(construtor)
lista.pop()
a = lista[3]+lista[4]
lista.reverse()
b = lista[4]+lista[5]
lista.append(lista[8])
c = len(lista)+lista[3]
d = a * b + c
```

13.5.1. Quais os valores das seguintes variáveis?

a =

b =

c =

d =

13.5.2. Ainda sobre o código acima, descreva quais os resultados dos cálculos abaixo?

print(3 * a + len(lista) * 2)

print(4 * b + 60 % c)

print(max(lista) * len(lista) // 2)

Lição 14. A Função range()

14.1. Conhecendo a função range()

A função **range()** cria um objeto "iterável" [uma sequência]. Para usar essa sequência como uma lista, converta-a por meio da função **list()** – como vimos na lição anterior.

***Dica:** um objeto "iterável" é um objeto que **pode ser criado por meio de um processo repetitivo [iterações]** e que pode ter cada um de seus itens acessados um a um em etapas. Uma sequência numérica construída pela função **range()** é considerada um iterável pela linguagem Python.*

A função range pode ser usada com um único argumento, como em

```
sequencia = range(10)
```

Neste caso, ele cria uma sequência de números inteiros que **começa em 0 [zero] e vai até 9 [nove]**. O número que é colocado como argumento é o "limite" da sequência. O limite da sequência **não é incluído** na sequência, por isso que a sequência acima foi apenas até o número 9.

A função range() também pode ser usada com mais de um argumento, como aqui:

```
sequencia = range(3,10)
```

Do jeito como essa função foi escrita, ela criará uma sequência de 3 até 9. Não esqueça que o número limite nunca é incluído na sequência, mas o número inicial sempre é. Nesta sequência, com início e fim, o primeiro argumento indica o início da sequência e o segundo número indica o número final [que nunca é incluído].

A função range() também pode ser escrita com 3 argumentos. O primeiro indica o número de início, o segundo indica o número final e o terceiro indica a razão [os passos] entre cada item. Ou seja, o código

```
sequencia = range(3,10,2)
```

vai trazer uma sequência que começa em 3, vai até 9 [10 não é incluído] e salta **de 2 em 2** a cada item, ou seja, a sequência será 3, 5, 7, 9.

Para resumir, a função range() cria uma sequência de números inteiros e pode ser escrita:

range(final): a sequência começa em **0 [zero]** e vai, de um em um, até **final-1**.

range(início, final): a sequência começa em **início** e vai, de um em um, até **final-1**.

range(início, final, passo): a sequência começa em **início** e vai, de **passo em passo**, até **final-1**.

É possível construir uma sequência decrescente [do maior para o menor]. Basta que o início seja maior que o final e que se escreva o passo negativo, como no exemplo:

```
>>> print(list(range(20,10,-2)))
```

```
[20, 18, 16, 14, 12]
```

14.2. "A Cara" da Função range()

A função range() é conhecida como "construtora". Ela serve para construir uma sequência imutável de números inteiros. Essa sequência **não é** uma lista, por isso não pode ser vista diretamente como uma lista. Vamos ver como a função range() aparece?

```
>>> a = range(3,20,4)
>>> print(a)
range(3,20,4)
```

Note que a variável a é mostrada como uma range, mesmo... não são apresentados os valores em si. Poderemos ver os valores se pedirmos para ver a variável convertida para uma lista:

```
>>> print(list(a))
[3, 7, 11, 15, 19]
```

14.3. Vamos Exercitar a Lição 14

14.3.1. Descreva quais são as listas criadas a partir destas range() :

list(range(20,30,3))

[, , ,]

list(range(9))

[]

list(range(25,5,-4))

[]

list(range(7,23,2))

[]

list(range(37,13,-5))

[]

list(range(10,30,5))

[]

Lição 15. A Estrutura de Repetição for

Na lição 12, fomos apresentados a inúmeros programas que exigiam várias vezes a mesma coisa [como pedir o nome ou a idade do usuário]. Naquele momento, fizemos vários exemplos escrevendo todas as vezes os mesmos comandos no código. Por exemplo: imagina se o programa tiver que pedir três vezes que o usuário digite os nomes dos alunos e depois adicionar esses nomes numa lista... como ficaria?

```
lista = []
nome = input("Digite o nome do aluno:") #Pedimos uma vez.
lista.append(nome) #Adicionamos uma vez.
nome = input("Digite o nome do aluno:") #Pedimos de novo.
lista.append(nome) #Adicionamos de novo.
nome = input("Digite o nome do aluno:") #Pedimos outra vez.
lista.append(nome) #Adicionamos outra vez.
```

É um saco, né?

Tem um jeito bem simples de fazer um [ou mais] comando ser repetido sem precisar escrever todo o código mais de uma vez. São as chamadas **estruturas de repetição** – ou **laços de repetição**. O primeiro laço de repetição que iremos conhecer é o laço **for**.

O laço for é usado quando nós **sabemos quantas vezes os comandos serão repetidos**, pois precisamos informar isso ao usá-lo.

Usamos o laço for da seguinte maneira:

```
for num in range(1,6):
    print(num)
```

Note que a segunda linha está um pouco afastada da margem esquerda. Esse efeito é chamado de **indentação** [reclamação]. Para a linguagem Python, a indentação é muito importante – Você simplesmente não pode errá-la, porque o programa simplesmente não funcionará!

A indentação informa que todo o código recuado **está dentro** do código anterior, ou seja, a linha **print(num)** está dentro do laço for num in **range(1,6)**.

Vamos entender o que significa este código?

for num in range(1,6) pode ser lido como **para cada <num> dentro da sequência 1 a 6 [sem o 6]...** ou seja, o Python vai:

- [a] Criar uma variável temporária chamada num [qualquer nome serve];
- [b] Atribuir à variável num o valor do primeiro item da sequência [no caso, o número 1];
- [c] Realizar os comandos dentro do laço [no caso, imprimir na tela o valor de num];

- [d] Atribuir à variável num o valor do próximo item da sequência;
- [e] Realizar os comandos dentro do laço para este valor;
- [f] Repetir os passos [d] e [e] até o final da sequência.

O resultado do código anterior é:

```
1
2
3
4
5
```

Podemos traduzir o programa acima em "português" [para que fique fácil entender] como:

```
Para cada num em range(1,6):
    mostre num na tela
```

A sintaxe [regra de escrita] correta do laço for é:

```
for variável in sequência:
    comandos a serem repetidos
    comandos a serem repetidos
    comandos a serem repetidos
Este comando já não faz mais parte do laço
```

Para indicar o final de um laço qualquer, é só escrever algum comando **sem a indentação** dos demais. Não esqueça, também, de colocar o sinal de : **[dois pontos]** no final do comando que estabelece o laço for.

Vamos ver outro exemplo de programa com o laço for:

```
for base in range(0,5):
    print("O quadrado de", base, "é", base ** 2)
    print("O cubo de", base, "é", base ** 3)
```

Este código vai repetir dois comandos print a cada passo da iteração. O resultado dele é:

```
0 quadrado de 0 é 0
0 cubo de 0 é 0
0 quadrado de 1 é 1
0 cubo de 1 é 1
0 quadrado de 2 é 4
0 cubo de 2 é 8
0 quadrado de 3 é 9
0 cubo de 3 é 27
0 quadrado de 4 é 16
0 cubo de 4 é 64
```

15.1. Usando o laço for com Listas

O laço for pode ser construído com listas também, desta forma, a variável temporária usada no comando é preenchida diretamente com os valores da lista, um a um, a cada passo.

Vamos ver um exemplo?

```
economistas = ['Mises', 'Hayek', 'Rothbard', 'Iorio', 'Barbieri']  
for nome in economistas:  
    print("Menos Marx, Mais", nome)
```

Este programa retorna:

```
Menos Marx, Mais Mises  
Menos Marx, Mais Hayek  
Menos Marx, Mais Rothbard  
Menos Marx, Mais Iorio  
Menos Marx, Mais Barbieri
```

15.2. Vamos Exercitar sobre a Lição 15

15.2.1. Construa um programa que imprima na tela o triplo de cada número de 1 a 10.

15.2.2. Construa um programa que imprima na tela todas as letras do alfabeto [cada letra sendo impressa numa linha diferente].

Lição 16. A Estrutura de Repetição while

Aprendemos, na lição passada, sobre a estrutura de repetição for. Ela é muito útil para conseguir repetir comandos quando sabemos exatamente o número de vezes que esses comandos serão repetidos.

Mas há alguns casos em que vamos ter que pedir para repetir sem saber quantas vezes temos que fazê-lo. Para esses casos, **o laço for não funciona**, temos que usar **o laço while** [significa "**enquanto**" em inglês].

A gente usa ele assim:

```
while proposição:
    faça isso
    faça isso
    faça isso
aqui não faz mais parte do laço while
```

Note que a parte principal do laço while é a sua condição – **uma proposição ser verdadeira**. Para que o laço while seja executado [ou seja, que os comandos dentro dele sejam executados], basta que a proposição lá em cima seja verdadeira.

Vamos analisar um código bem simples:

```
contador = 1
while contador <= 5:
    print(contador)
    contador += 1
```

Neste programa:

- [a] Começamos definindo que a variável contador tem valor 1.
- [b] Estabelecemos que enquanto contador for menor ou igual a 5, os comandos abaixo serão repetidos.
- [c] O programa irá imprimir [na tela] o valor atual da variável contador.
- [d] O programa irá somar 1 à variável contador [explicação abaixo].
- [e] O programa volta a analisar a condição do while lá em cima, se ela continuar verdadeira, o programa executará os comandos internos ao while novamente.

Atenção: a expressão **contador += 1** é uma forma reduzida de escrever **contador = contador + 1**.

Atenção 2: é necessário colocar algum comando no laço while para atualizar a variável analisada na condição. Sem esse comando, o valor da variável contador será sempre 1 [conforme definido lá em cima], o que **fará o while ser eterno** [repetir eternamente os comandos porque a condição será sempre verdadeira, o que chamamos de **Loop Infinito**].

O resultado do programa mostrado acima é:

```
1
2
3
4
5
```

16.1. Loop Infinito

Quando é criado um laço de repetição que não tem como parar, ou seja, que vai se repetir para sempre, chamamos isso de **Loop Infinito**. Às vezes, o Loop Infinito é um erro, uma falha... e, portanto, deve ser corrigido. Outras vezes, porém, é necessário fazer um loop infinito que ofereça alguma forma de sair dele em caso de emergência.

Vamos avaliar um pequeno código com erro de loop infinito:

```
senha = 1
while senha <= 5:
    print("a senha é:", senha)
```

Note que em nenhum momento foi incluído um comando que altera o valor da variável `senha`. **Ela ficará sempre com valor 1**. Como o valor 1 é menor ou igual a 5, então a proposição `senha <= 5` será sempre verdadeira. Esse while não terá fim... ele vai repetir eternamente o comando `print`, resultando em:

```
a senha é: 1
a senha é: 1
a senha é: 1
a senha é: 1
a senha é: 1
```

E assim por diante... pra sempre! Se não é o seu desejo que isso aconteça, nunca esqueça de incluir a cláusula que altera o valor da variável que é usada como condição do laço `while`.

16.2. A cláusula else

O laço `while` repetirá comandos enquanto a sua condição [proposição] for verdadeira. Mas e quando não for mais? Bom, quando uma condição deixa de ser verdadeira, o programa simplesmente deixa de repetir o laço e passa a executar o primeiro comando após o laço [fora dele].

Existe a possibilidade de adicionar uma cláusula chamada `else` [senão, em inglês] que faz o Python obedecer quando a condição do laço `while` não for mais verdadeira. Veja o código a seguir:

```
a = 1
```

```
while a < 4:
    print(a)
    a += 1
else:
    print("acabou a repetição")
print("fim do programa")
```

O resultado do programa será:

```
1
2
3
acabou a repetição
fim do programa
```

No final das contas, a cláusula else nem é tão importante assim porque quando a condição da while for falsa, o laço deixará de ser executado [repetido] e o próximo comando [imediatamente abaixo e fora do while] já vai ser executado.

16.3. *while True*

Uma forma mais fácil de fazer um Loop Infinito [se necessário] é usar o laço **while** com a condição **True**. Lembre-se de que **True** é um **valor booleano e significa Verdadeiro**, portanto, é imutável, será considerado sempre verdadeiro.

```
while True:
    print("sempre")
```

Esse código simples vai imprimir a palavra "sempre" infinitas vezes porque a condição do while é **True**.

16.4. *Vamos Exercitar a Lição 16?*

16.4.1. Construa um programa que solicite vários nomes dos usuários e os guarde numa lista até que o usuário digite "fim". Depois de finalizar a captura dos nomes, o programa irá exibir a lista contendo todos os nomes inseridos pelo usuário.

16.4.2. Construa um programa que solicite diversos preços ao usuário. O programa deve entender que o número 0 [zero] indicará o fim da captura dos dados. Ao término da captura de todos os números, o programa deverá realizar a soma de todos os preços capturados.

Tens Instagram? Me segue lá! [*@ProfessorJoaoAntonio*](#).

Lição 17. Estrutura Condicional if

17.1. Conceitos básicos do if

Algumas vezes, é necessário fazer com que o próprio programa "decida" por seguir um caminho ou outro. Este caso é resolvido com uma estrutura muito conhecida chamada **if** ("**se**", em inglês). A cláusula **if** é usada assim:

```
if proposição:
    faça isso
    faça isso
    faça isso
aqui já não faz mais parte do if
```

Novamente, de forma bem semelhante ao laço **while**, a condição para que o **if** aconteça é uma proposição ser verdadeira. Se ela **for** verdadeira, **todos os comandos dentro do if serão obedecidos uma vez**. Se a proposição **não for** verdadeira, **nenhum comando dentro do if será executado** [ou seja, o bloco inteiro será ignorado].

Vejamos:

```
senha = 20
if senha > 10: # SE a variável senha for maior que 10 [e ela é]...
    print("A senha é:", senha) # ...este código será executado
print("Fim do Programa")
```

O resultado deste código é:

```
A senha é: 20
Fim do Programa
```

Note o código abaixo, em que a variável **senha** não é maior que 10 inicialmente:

```
senha = 5
if senha > 10:
    print("A senha é:", senha) #Este código será ignorado
print("Fim do Programa")
```

O resultado deste código é:

```
Fim do Programa
```

17.2. Cláusula else

A estrutura **if** também tem a sua cláusula **else**, e funciona melhor aqui. Os comandos presentes dentro do bloco **else** **só serão executados caso a condição do if seja falsa**. Caso a condição da **if** seja verdadeira, os comandos dentro da **else** não serão executados.

Vamos a um exemplo:

```
senha = 5
if senha > 10: #Esta condição é FALSA
    print("A senha é:", senha) #Este código será ignorado
else:
    print("A senha é menor que 10!") #Este código será executado
print("Fim do Programa")
```

O resultado deste código é:

```
A senha é menor que 10!
Fim do Programa
```

Note que o bloco dentro do if foi ignorado e apenas o bloco dentro do else foi executado.

17.3. Cláusula elif

A cláusula **elif** é usada quando você precisa **fazer outro teste** [além do teste do if] antes de partir para o uso de uma cláusula else.

Um exemplo que sempre usam para ilustrar isso é sobre uma nota escolar. Vamos supor que a regra é essa: [a] se tirar nota menor que 5, está "reprovado". [b] se não tirar menos que 5, mas tirar menos que 7, vai para a "prova final". [c] se tirar 7 ou mais, ele estará automaticamente aprovado.

*Dica: vale lembrar que a cláusula **elif** é uma contração de **else if** ("**senão, se**", em inglês).*

O código a seguir mostra como seria feito um programa para a regra acima:

```
nota = float(input("Digite a nota do aluno:"))
if nota < 5:
    print("O aluno está REPROVADO")
elif nota < 7:
    print("O aluno vai para a PROVA FINAL")
else:
    print("O aluno está APROVADO")
print("Fim do Programa")
```

Perceba que **if**, **elif** e **else** devem ter exatamente a mesma indentação [ou seja, o mesmo afastamento da margem do editor].

Explicando o código acima:

- [a] Pedese que o usuário digite a nota do aluno [função **input()**] e converte-se o que ele digitar em um **número float** [função **float()**], armazenando isso na **variável nota**.
- [b] Inicia-se o teste **if** para saber **se nota é menor que 5**. Caso isso seja verdade, apresenta-se na tela a frase **"O aluno está REPROVADO"**.

- [c] Caso este teste anterior seja falso [ou seja, a nota não é menor que 5], inicia-se outro teste [**elif**], agora para saber **se a nota é menor que 7**. Caso isso seja verdade, apresenta-se na tela a mensagem **"O aluno vai para a PROVA FINAL"**.
- [d] Caso o teste anterior também seja falso [ou seja, se a nota não for menor que 7], resta obedecer à cláusula **else**, apresentando na tela a mensagem **"O aluno está APROVADO"**.

17.4. Vamos Exercitar a Lição 17?

- 17.4.1. Escreva um programa que receba vários nomes e idades de usuários e os adicionem a uma lista caso eles sejam maiores de idade [18 anos ou mais velho] e os adicionem a outra lista caso eles sejam menor de idade [menor de 18 anos].

Lição 18. Caracteres Especiais nas Strings

18.1. Por que Caracteres Especiais?

Em alguns casos, podemos precisar apresentar sinais [símbolos] especiais nos strings que apresentamos na tela para o usuário. Um exemplo simples é o caractere das aspas. Como poderíamos apresentar o caractere das aspas para os usuários? Que tal assim:

```
print("Que tal apresentar "aspas" para o usuário?")
```

Note que a linha acima apresentará um problema, porque a segunda " será entendida como sendo o fechamento da primeira... Isso porque a função print() a entenderá assim. Ela não será apresentada como eu gostaria que fosse:

```
Que tal apresentar "aspas" para o usuário?
```

Se você quer, realmente, mostrar aspas no texto que aparecerá para o usuário, você deve usar um código interessante no Python... esse aqui:

```
print("Que tal apresentar \"aspas\" para o usuário?")
```

O símbolo de \" [barra-invertida seguido de aspas] é o indicador de um caractere especial. Ele é a forma de dizer ao Python que as aspas que seguem a barra-invertida devem ser apresentadas exatamente daquele jeito na saída – ou seja, as aspas não serão consideradas parte do código do print(), e sim, **aspas dentro do texto**. Há outros exemplos de caracteres especiais que usam a barra-invertida.

A combinação \' [barra-invertida com aspas simples] é usada para apresentar uma aspas simples [também chamada de apóstrofo] no texto.

```
print("Vamos apresentar um \'apóstrofo\' no texto")
```

resulta em:

```
Vamos apresentar um 'apóstrofo' no texto
```

A combinação \t [barra-invertida com a letra t] serve para colocar uma **tabulação horizontal** naquela posição na linha [que é aquele espaço criado pelo uso da tecla TAB].

```
print("NOME \t FONE \t IDADE")  
print("Joao \t 9876-5432 \t 42")
```

resulta em:

NOME	FONE	IDADE
Joao	9876-5432	42

A combinação \t é usada para a organização visual dos dados na tela do usuário. É possível colocar mais de um \t seguidamente, para aumentar os espaços entre os dados. Seria usando \t\t\t, por exemplo.

A combinação `\n` [barra-invertida seguida da letra n] é usada para criar **uma nova linha** naquela posição. Seria o equivalente ao digitar ENTER num texto num programa qualquer.

```
print("Continuando... \nEra uma vez... \nUm sapo barbudo...")
```

resulta em:

```
Continuando...
Era uma vez...
Um sapo barbudo...
```

A combinação `\v` [barra-invertida seguida da letra v] serve para inserir uma **tabulação vertical** nessa posição do texto. Uma tabulação vertical é como uma nova linha, mas **"com mais espaço entre as linhas"** [aumenta a distância entre as linhas com o `\v`].

```
print("Nove dedinhos\vForam Passear\vAlém de Atibaia para brincar...")
```

resulta em:

```
Nove dedinhos

Foram Passear

Além de Atibaia para brincar...
```

Ei, João, e se eu quiser apresentar ao usuário a própria barra-invertida?

Aí você usará, caro leitor, cara leitora, a combinação `\\` [duas barras-invertidas] exatamente no local onde deseja que a barra-invertida apareça para o usuário.

```
print("Mostrando uma \\ [barra-invertida] para o usuário")
```

resulta em

```
Mostrando uma \ [barra-invertida] para o usuário
```

18.2. Aspas Triplas

Já havíamos sido apresentados rapidamente às aspas triplas, não é mesmo? O recurso de `"""` [aspas triplas] pode ser usado para fazer comentários no meio do código, como já havíamos visto, mas também pode ser usado para escrever um conteúdo no corpo do código exatamente como a gente quer que ele seja escrito para o usuário [inclusive podendo usar códigos especiais como os que vimos acima].

Tudo aquilo que estiver dentro das aspas triplas será respeitado exatamente como estiver lá, incluindo as quebras de linha que você usou [ENTER] no meio do código. Vamos ver isso?

```
texto = """
Quando você perceber que, para produzir, precisa obter a autorização de
```

```

quem não produz nada; quando comprovar que o dinheiro "flui" para quem
negocia não com bens, mas com favores; quando perceber que muitos ficam
ricos pelo suborno e por influência, mais que pelo trabalho, e que as leis
não nos protegem deles, mas, pelo contrário, são eles que estão protegidos
de você; quando perceber que a corrupção é 'recompensada', e a honestidade
se converte em auto-sacrifício; então poderá afirmar, sem temor de errar,
que sua "sociedade" está condenada. \t Ayn Rand"""
print("*" * 73)
print(texto)
print("*" * 73)

```

O resultado deste programa é:

```

*****
Quando você perceber que, para produzir, precisa obter a autorização de
quem não produz nada; quando comprovar que o dinheiro "flui" para quem
negocia não com bens, mas com favores; quando perceber que muitos ficam
ricos pelo suborno e por influência, mais que pelo trabalho, e que as leis
não nos protegem deles, mas, pelo contrário, são eles que estão protegidos
de você; quando perceber que a corrupção é 'recompensada', e a honestidade
se converte em auto-sacrifício; então poderá afirmar, sem temor de errar,
que sua "sociedade" está condenada.                Ayn Rand
*****

```

Perceba que na penúltima linha foi usado um `\t`. Este código especial foi respeitado e se converteu numa tabulação na saída para o usuário, antes do nome da **Ayn Rand**. Perceba também que as **aspas [duplas e simples]** nas palavras "flui", 'recompensada' e "sociedade" foram inseridas sem que fosse necessário usar códigos especiais para sua inserção.

Nas aspas triplas, tais caracteres e as quebras de linha não precisam ser apresentadas com caracteres especiais `\`. Apenas a tabulação horizontal `[\t]` e a tabulação vertical `[\v]` é que precisam ser oficialmente apresentadas.

Ahhh, também não custa lembrar que tanto faz se você usa `"""` **[aspas duplas triplas]** ou `'''` **[aspas simples triplas]**, pois o resultado é exatamente o mesmo!

18.3. Vamos Exercitar a Lição 18?

18.3.1. Quais os resultados dos códigos abaixo?

```

a = " A civilização é o avanço de uma sociedade\nem direção à privacidade.
O selvagem tem uma vida pública,\nregida pelas leis de sua tribo.\n
Civilização é o processo de libertar\nno homem dos outros homens.\v\tAyn
Rand"
print(a)

```

```
a = """Só existem duas formas dos homens lidarem uns com os outros:  
Armas ou lógica. \t\t Força ou persuasão.  
Aqueles que sabem que não podem vencer por meio da lógica sempre recorrem  
às armas.\vAyn Rand"""  
print(a)
```

Lição 19. Formatando Texto

19.1. Uma Nova Forma de Concatenar Textos e Variáveis

Vamos analisar o código a seguir:

```
nome = "João"
idade = 42
print(Nome + " tem " + str(idade) + " anos de idade.")
```

O resultado deste programa é a impressão, para o usuário da frase:

```
João tem 42 anos de idade.
```

Bom, isso foi um jeito simples, e já conhecido, de concatenar, numa única string, algumas variáveis e algumas strings. O uso do sinal de + [adição] para concatenar já é conhecido, né?

Lembrou que precisamos colocar um espaço antes e/ou depois de cada trecho de texto entre aspas, né? Se não fizermos isso, as palavras concatenadas ficarão todas juntas.

Perceba que, neste caso [usando o sinal de +], também foi necessário usar a função `str()` na variável `idade`, porque ela é do tipo `integer`, conforme vemos lá em cima no momento em que nós atribuímos a ela o valor 42.

*Lembre-se que **não dá para concatenar** uma string e um valor não-string com o sinal de +.*

Peraí, João, mas podemos escrever esse mesmo programa de outra forma, não podemos?

Sim, amigo leitor, amiga leitora! Podemos escrever o mesmo comando `print()` de outra maneira! Podemos separar cada item a ser impresso por uma vírgula. Neste caso, nem precisamos converter os valores inteiros [ou seja, não precisaremos da função `str()`] e nem precisaremos colocar os espaços antes e depois de cada trecho de string:

```
nome = "João"
idade = 42
print(Nome, "tem", idade, "anos de idade.")
```

O resultado deste programa é o mesmo:

```
João tem 42 anos de idade.
```

Estas são facilidades da função `print()` – tratar números como strings e colocar automaticamente um espaço entre os itens. Espertinha, não?

Agora só resta conhecer uma outra forma de trabalhar "misturando" textos e variáveis: o método `.format()`.

O método **.format()** pode ser usado com qualquer string e serve para colocar, nas posições certas, os valores que forem apontados dentro dos parênteses. Você escreve ele assim: **string.format()**.

O funcionamento é um pouco mais complicado que os exemplos anteriores, mas não é nada impossível de entender! Vamos usar **{}** [chaves] para definir o exato local onde os dados devem aparecer dentro da string. Veja o exemplo:

```
nome = "João"
prof = "professor"
mensagem = "O amigo {} atua como {} em nossa escola".format(nome, prof)
print(mensagem)
```

Note que os indicadores **{}** são colocados onde desejamos que os dados sejam incluídos. No exemplo acima, colocamos **dois** desses indicadores: um depois de "amigo" e o outro depois de "como". O método **.format()** foi preenchido com dois argumentos: **nome** e **prof** [as duas variáveis que havíamos criado].

Como há dois indicadores **{}** e temos dois argumentos dentro do método **.format()**, cada um deles será substituído por cada argumento do método, em ordem de ocorrência. Ou seja, **o primeiro {}** será substituído pelo argumento **nome** e o **segundo {}** será substituído pelo argumento **prof**.

Uma outra forma, melhorada, permite que você use combinações mais poderosas com esse recurso é que você pode colocar nomes nas chaves, fazendo com que possa apontar para eles mais de uma vez. Veja no exemplo:

```
num = 12
nome = "Mises"
mensagem = "{a}. Nosso autor número {a} é o {b}.".format(a=num, b=nome)
print(mensagem)
```

O resultado deste código é:

```
12. Nosso autor número 12 é o Mises.
```

Note que é necessário:

- [a] Definir, dentro das chaves, algum **apelido** para o dado que será ali colocado;
- [b] Dentro dos parênteses, escrever a expressão **apelido=variável** para associar o apelido usado na string com alguma variável já existente. É necessário nomear cada apelido que será usado na string. Não funcionará se você se esquecer de algum.

Perceba que neste caso do **.format()**, também não é necessário converter valores que são originalmente números [como foi o caso da variável **num**].

19.2. Vamos exercitar? Agora sobre a Lição 19!

19.2.1. Descreva qual a saída [na tela] do programa a seguir:

```
lista = ['Mises', 'Hayek', 'Menger', 'Hoppe', 'Friedman']
for index in range(1,4):
    mensagem = "{num}. {nome}"
    print(mensagem.format(num=index, nome=lista[index]))
print("Lista encerrada")
```


Lição 20. Muitos Programas para Estudar!

20.1. Descreva quais as saídas [na tela] para os Programas a seguir:

[Responda diretamente aqui – sem usar o Python – use a sua mente para responder]

20.1.1. Programa 1 – Trabalhando com Listas

```
nomes = ['Mises', 'Hayek', 'Hoppe', 'Friedman']
for xy in nomes:
    if xy != "Hayek":
        print("O autor é {}".format(xy))
    else:
        print("{} é um Gênio!")
print("Fim do Programa")
```

20.1.2. Programa 2 – Cálculos

```
a, b, c = 10, 20, 30
if c > 3 * b:
    a += 10
    b *= a
else:
    a *= c
    b += a
print(a, b)
```

20.2. Escreva um programa que...

20.2.1. Receba nomes e notas dos alunos e os insira em duas diferentes listas. Ao digitar "x", o sistema para de capturar nomes e notas e já mostra a lista dos alunos no seguinte formato:

```
=====
Nome          Nota
=====
Fulano        9.0
Beltrano      2.0
Trajano       5.0
```

20.2.2.Receba vários números diferentes até que se digite "x". Depois de encerrada a captura dos números, o sistema irá apresentar na tela: [a] a lista contendo todos os números inseridos pelo usuário apresentados em ordem crescente, [b] o maior número entre eles, [c] o menor número entre eles, [d] a soma de todos os números inseridos e [e] a média aritmética de todos eles.

20.2.3.Receba vários nomes e notas dos alunos [encerra a recepção dos dados quando digitar "x"]. Se a nota do aluno for menor que 7, que o nome dele e a sua nota sejam armazenados numa determinada lista – a lista de alunos problemáticos. Se a nota do aluno for igual ou maior que 7, que seu nome e sua nota sejam inseridos em outra lista – a lista de bons alunos. Ao término do programa, mostrar a seguinte tela:

```
=====
Alunos Bons
=====
Fulano          9.0
Beltrano        8.0
Trajano         7.5
=====
Alunos Problemáticos
=====
Mariano         3.0
Lugano          5.0
```

Lição 21. Construindo Nossas Funções

Uma **função** é um "pedaço de código", um "trecho" de código, um "fragmento" de código que pode ser usado mais de uma vez! Nós construímos funções para não termos que repetir código desnecessariamente.

Podemos criar funções usando o comando **def**, que é uma contração para "definir". Ou seja, o ato de **definir uma função** é o ato de criá-la, de dizer como ela vai funcionar, quais as formas de usá-la.

Pense numa função como uma ferramenta [um martelo ou um alicate, por exemplo]. Se você não tem um martelo, mas pode construí-lo, **faça-o**! A partir daquele momento, você vai usar sempre aquele martelo! Ele será muito útil! Ou seja, **você o constrói UMA VEZ**, mas depois disso **poderá usá-lo VÁRIAS VEZES**!

Vamos construir nossa primeira função:

```
def rodape():  
    print("=" * 30)  
    print("Atenção! O programa terminou!")  
    print("=" * 30)
```

Pronto! Criamos uma função que será usada sempre no final de nossos programas, para colocar na tela o seguinte:

```
=====  
Atenção! O programa terminou!  
=====
```

Uma vez criada a função, sempre que precisarmos usá-la, nós a **chamamos** [o termo que os vários autores de informática usam é esse: "**chamar uma função**" – é o mesmo que "mencionar seu nome", ou "**usar aquela função**"].

Basta, realmente, mencionar aquela função no exato local do código onde você quer que ela execute seus comandos. Olha isso:

```
def rodape():      #Aqui, a função foi definida [ou seja, criada]  
    print("=" * 30)  
    print("Atenção! O programa terminou!")  
    print("=" * 30)  
num = 80  
print(num * 3)  
rodape()          #Aqui, a função foi chamada [ou seja, utilizada]
```

O resultado deste código pode ser visto na página seguinte:

```
240
=====
Atenção! O programa terminou!
=====
```

Não esqueça que a criação de uma função exige, também, que todos os comandos dentro daquela função sejam indentados [recuados] para que o Python entenda que eles fazem parte da função, como mostra o exemplo a seguir:

```
def funcao_exemplo():
    este comando faz parte da função
    este comando faz parte da função
este comando NÃO FAZ PARTE da função
```

21.1. Construindo Funções com Argumentos

Além de serem usadas para "repetir" trechos de código ao longo do programa, Funções podem ser usadas para tratar algum valor que é informado a elas.

No caso mostrado anteriormente [da função `rodape()` que nós havíamos criado], a função simplesmente não recebe informação alguma. Ela simplesmente **é chamada e realiza os comandos** que haviam sido colocados dentro dela.

Podemos, porém, criar funções que recebam dados [os argumentos] e que, com isso, façam alguma coisa com esses dados. Vamos ver o exemplo a seguir:

```
def dobro(num):    #Aqui, a função é definida para receber um argumento
    print("O dobro de", num, "é", num*2)
print("Testando a Função com argumentos")
dobro(15)
dobro(20)
dobro(80)
```

O resultado deste código é:

```
Testando a Função com argumentos
O dobro de 15 é 30
O dobro de 20 é 40
O dobro de 80 é 160
```

Note que a cada vez que a função `dobro()` foi chamada, nós colocamos um número diferente dentro dos parênteses [argumento]. A cada novo argumento **passado** para a função, ela realizou seu comando "print" com um dado diferente, imprimindo uma informação diferente.

Então, vamos a um "vocabulário" para definir alguns conceitos sobre funções:

Definir uma Função ou "Declarar" uma função: significa dizer ao Python que aquela função existe, ou seja, é definir uma função é construí-la.

Chamar uma Função: é a mesma coisa que usá-la. Chamar uma função é mencionar o seu nome no meio do programa. É assim que nós usamos as funções.

Passar um Argumento para uma Função: é a mesma coisa que informar um dado para que a função trabalhe corretamente. Algumas funções não pedem argumentos, outras, porém, necessitam de dados para trabalhar... passar um argumento [ou parâmetro] é justamente colocar um dado dentro dos parênteses de uma função para ela trabalhar com ele.

21.1.1. Funções que recebem vários argumentos

Uma função pode receber mais de um argumento, desde que você a declare assim. Vejamos o exemplo abaixo:

```
def maior(a,b):  
    if a > b:  
        print(a, "é maior que", b)  
    elif b > a:  
        print(b, "é maior que", a)  
    else:  
        print("os números são iguais")  
print("Testando Função com dois argumentos")  
maior(10,30)  
maior(80,30)  
maior(30,30)
```

O resultado do programa acima é o seguinte:

```
Testando Função com dois argumentos  
30 é maior que 10  
80 é maior que 30  
os números são iguais
```

21.2. Vamos Exercitar a Lição 21?

21.2.1. Construa uma função que receba dois valores e que apresente na tela a soma deles. Depois, chame-a com os valores 30 e 50.

21.2.2. Construa uma função que receba três valores e que imprima na tela uma mensagem informando se há valores repetidos entre eles. Chame a função passando os valores 30, 20 e 30 e depois chame-a com os valores 10, 30 e 80.

21.2.3. Construa uma função que receba três palavras e as imprima na tela em sequência alfabética, com um espaço entre elas.

Lição 22. As Variáveis Dentro das Funções

22.1. Analisando o "Escopo" de uma Variável

Em programação, é muito importante entender o "escopo" das variáveis. Uma variável tem uma "área de atuação", uma "região" onde ela atua... isso é o **escopo** dela.

Por exemplo, **uma variável que foi definida dentro de uma função só funcionará dentro daquela função**, mesmo que haja outra variável com o mesmo nome sendo usada fora da função.

Vejamos o programa a seguir:

```
def calculo(a, b, c):  
    print(a + b * c)  
calculo(2,3,4)  
teste = a + 10    #Aqui vai dar erro porque não há variável a conhecida
```

No programa acima, as variáveis **a**, **b** e **c** foram definidas dentro da função **calculo()**. Elas não podem ser usadas fora dela. A linha **teste = a + 10** vai apresentar um erro porque a variável **a** nunca foi declarada antes no programa.

Outra coisa é que **as variáveis declaradas fora da função não influenciam as variáveis declaradas dentro da função**, mesmo que tenham o mesmo nome – para o Python, são variáveis diferentes.

```
a = 10    #Essas variáveis não influenciam as variáveis na função  
b = 4  
def calculo(a, b, c):    # Essas variáveis a, b e c só existem aqui  
    print(a + b * c)    # dentro dessa função  
calculo(b,b,b)    # Aqui, estamos chamando a função e passando os  
calculo(a,a,a)    # parâmetros [argumentos] que queremos para ela  
calculo(a,b,10)
```

O resultado do programa acima vai ser a impressão de três linhas na tela, contendo os valores:

```
20  
110  
50
```

22.2. O Valor Padrão de uma Variável

Se uma função foi construída para receber 3 argumentos, o usuário deverá chamá-la no programa informando esses 3 argumentos [essa é a regra geral]. Se você constrói uma função qualquer com:

```
def funcao_qualquer(arg1, arg2, arg3):
```

Mas na hora de chamar a função, você faz a besteira de chamá-la assim:

```
funcao_qualquer(100, 200)
```

[note que ela foi chamada informando apenas dois argumentos] – Isso vai dar um erro! O Python irá mostrar uma mensagem de erro:

```
File "/Users/joao/Documents/prova.py", line 8, in <module>
    funcao_qualquer(100,200)
TypeError: funcao_qualquer () missing 1 required argument: 'arg3'
```

A última linha da mensagem diz **"missing 1 required argument"** [1 argumento necessário está faltando], e aponta que é o **arg3** que está em falta.

Para evitar isso, e, claro, para tornar um dos argumentos desnecessário, podemos criar um **"valor padrão"** para o argumento, caso ele não seja informado.

É muito fácil fazer isso: basta colocar **argumento=valor** na própria definição da função. Veja:

```
def maior(a,b=10):      #Aqui, definimos que 10 é o valor-padrão de b
    if a > b:
        print(a, "é maior que", b)
    elif b > a:
        print(b, "é maior que", a)
    else:
        print("os números são iguais")
print("Testando Função com dois argumentos")
maior(10,30)            #Aqui o 30 é passado como argumento b
maior(80)               #Aqui o 10 [valor-padrão] será considerado como b
maior(6)               #Aqui também o valor-padrão será considerado como b
```

O programa vai apresentar na tela isso aqui:

```
Testando Função com dois argumentos
30 é maior que 10
80 é maior que 10
10 é maior que 6
```

Perceba que nos casos em que não informamos o segundo argumento, o Python considerou que o argumento b da sua função tem valor 10. Isso evita o erro e permite que você crie um valor básico, comum, para o argumento. Você vai perceber a necessidade disso em algum momento enquanto estiver programando! Todos os argumentos de uma função podem ter valor-padrão, veja:

```
def soma(num1=30,num2=20):
    print("A soma dos dois é:", num1 + num2)
soma(10,400)
soma(80)
soma()
```

O resultado deste código é:

```
A soma dos dois é: 410  
A soma dos dois é: 100  
A soma dos dois é: 50
```

Note que no primeiro exemplo, **foram passados os dois argumentos** na função. Ela resultou, portanto, em 410 [a soma do 10 com o 400]. No segundo exemplo, **só foi passado o argumento 80**, o Python considerou o segundo argumento com seu valor-padrão [que é 20], portanto, o resultado deu 100.

No terceiro exemplo, **nenhum argumento foi passado** para a função – ela teve de considerar os dois valores-padrão: 30 e 20. Por isso o resultado deu 50.

22.3. Vamos Exercitar a Lição 22?

22.3.1. Qual a saída do programa a seguir?

```
num = 30  
def calculo(num, pot=2):  
    if num <=100:  
        print("{} elevado a {} é igual a {}".format(num, pot, num**pot))  
    else:  
        print("Não é possível fazer cálculos maiores que", 100**2)  
calculo(20)  
calculo(2,5)  
calculo(120,130)
```

Lição 23. Funções Retornando Valores

23.1. Funções podem Trazer Resultados

Até agora, usamos funções apenas para "mostrar" valores [colocamos um comando `print()` dentro delas para que elas possam apresentar valores na tela]. Acontece que uma função pode ser escrita para não mostrar valores, mas para **retorná-los** [obter resultados] e nós vamos ver isso agora.

Acompanhe o programa simples a seguir:

```
def mostra_dobro(num):  
    print(num*2)  
mostra_dobro(10)
```

A função **`mostra_dobro()`** exige um argumento como valor e vai **mostrar o valor na tela** [graças à função `print()` lá dentro]. Por causa disso, a função pode ser usada diretamente como na linha **`mostra_dobro(10)`**. O resultado do programa acima é:

```
20
```

O número 20 foi impresso na tela porque a função **`print(num*2)`** está dentro da função **`mostra_dobro(num)`** – e nós chamamos esta função passando para ela o valor **10**.

Vamos aprender uma nova forma de utilização das funções? Olha o programa abaixo:

```
def calcula_dobro(num):  
    return num * 2
```

Note que a função **`calcula_dobro()`** não manda, em seu interior, nenhuma informação ser impressa na tela [não existe a função `print()` ali dentro]. O comando **`return`** indica que a função deve oferecer como resultado [ou seja, como resposta da função] justamente o **`num * 2`**.

Mostrar na tela é uma coisa [consegue-se isso com a função **`print()`**]. Retornar um valor é outra coisa bem diferente! Quando uma função retorna um valor, a gente tem que fazer algo com esse valor: que pode ser **armazená-lo numa variável** ou mesmo **imprimi-lo diretamente** por meio de uma função `print()` no corpo do programa. Veja isso:

```
def calcula_dobro(num):  
    return num * 2  
numero = calcula_dobro(10)  
print(numero)
```

No programa acima, a variável **`numero`** receberá o resultado da função **`calcula_dobro()`** aplicada ao valor **10**. Depois, é só mandar imprimir o valor da variável **`numero`**.

23.1.1. Os Segredos do Comando return

Dentro do código de uma função, o comando que indica qual será o valor-resultado desta função é o **return**. Nada colocado após o comando return no corpo da função será entendido. Tudo será ignorado.

```
def ignora(numero):  
    return numero * 10      #A função termina aqui  
    print("pronto")  
    numero += 20  
    print("numero")  
ignora(5)  
print("Programa Terminado")
```

O programa acima resultará, apenas, em:

```
Programa Terminado
```

Note que **nenhum** dos comandos **print()** dentro do corpo da função **ignora()** foi executado. Isso porque o código da função só é executado até o comando **return**. **Tudo aquilo que vier depois do comando return é, simplesmente, ignorado.**

O comando return pode ser escrito da maneira como vimos acima:

```
return a + b
```

ou como se fosse uma função:

```
return(a + b)
```

O Python entende as duas formas.

23.2. Função Retornando Mais de Um Valor

Uma Função pode retornar mais de um valor – e esses valores podem ser atribuídos a mais de uma variável ao mesmo tempo... Veja o exemplo:

```
def dobro_triplo(ancora):  
    return ancora*2, ancora*3  
num1, num2 = dobro_triplo(10)  
print(num1, num2)
```

O resultado deste programa é:

```
20    30
```

Note que a função **dobro_triplo()** foi construída para retornar dois valores: **ancora*2** e **ancora*3** [ancora é o nome da variável interna que será passada como argumento da função]. O fato de a função "devolver" dois valores ao mesmo tempo permite que usemos isso para preencher duas variáveis diferentes ao mesmo tempo, o que fizemos na atribuição das variáveis **num1** e **num2**.

23.3. Que Tal Exercitar a Lição 23?

23.3.1. Quais as saídas apresentadas pelos programas a seguir?

```
def media(num1, num2):  
    return (num1 + num2) / 2  
media(10,20)  
media(30,10)  
print("Fim do Programa")
```

```
def maior(num1, num2):  
    if num1 > num2:  
        return(num1)  
    else:  
        return(num2)  
a = 20  
b = 30  
grande = maior(a,b)  
print("O maior número entre", a, "e", b, "é", grande)
```

```
def teste(a, b, c=5):  
    return a + c * b  
hoje = teste(10,20,2)  
ontem = teste(20,10)  
print(hoje + ontem)  
print(teste(30,5,5))
```

Lição 24. List Comprehensions

24.1. O que são as List Comprehensions?

O Python tem uma forma muito fácil de criar listas seguindo as normas de uma expressão lógica. **List Comprehensions** seriam, justamente, essas **expressões lógicas**.

***Podemos definir assim:** List Comprehensions são expressões lógicas que servem de regra na criação de listas, determinando quais elementos vão fazer parte da lista e quais não irão.*

Não usamos o termo "List Comprehension" de forma traduzida para o português porque fica realmente bem estranho... seria "Compreensões da Lista" – ou talvez seria bom traduzir como "Lista Compreensível" [que meigo, não?].

Vamos entender como funciona? Primeiro, vamos imaginar que queremos criar uma lista com todos os quadrados dos números de 1 a 10. A lista desejada ficará exatamente assim:

```
lista = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100].
```

Note que 1 é 1^2 , 4 é 2^2 , 9 é 3^2 , 16 é 4^2 e assim por diante. A lista obedece aos critérios exigidos mencionados anteriormente, mas foi construída manualmente, ou seja, o programador teve de digitar cada item um a um. Um trabalho manual e muito chato [imagina se fossem 100 itens...].

Dá para construí-la de forma automática? Sem recorrer a fazer todos os itens manualmente? Sim! Dá uma olhada na expressão abaixo:

```
lista = [x**2 for x in range(1,11)]
```

Traduza essa expressão como: **crie uma lista com item de valor x^2 para cada x dentro da sequência range(1,11)**. [lembre-se que a range() não inclui o item final, ou seja, o 11].

O Python, portanto, vai iterar x dentro da range(1,11) [isso significa que x vai variar de 1 a 10] e a cada iteração, vai elevar o valor de x ao quadrado [é o que significa o x^2 , lembra?] e inserir esse valor na nova lista.

Vamos experimentar outro exemplo?

```
lista = [num+5 for num in range(10,200,13)]
```

Esta expressão significa: **para cada item num na sequência que vai de 10 a 200, saltando de 13 em 13, crie um item de valor num + 5**. O resultado deste comando é uma lista com os seguintes itens:

```
[15, 28, 41, 54, 67, 80, 93, 106, 119, 132, 145, 158, 171, 184, 197]
```

24.1.1. Usando if na Expressão para ser mais seletivo

As List Comprehensions podem ganhar uma ajuda incrível da cláusula **if**. Sim! Podemos ser bem mais seletivos e escolher alguns itens apenas da listagem a ser criada [ou seja, não aceitar todos eles, ou definir mais uma exigência para a inclusão dos itens na nova lista].

Que tal se a regra fosse: **adicione todos os números de 1 a 30 apenas se eles forem múltiplos de 4.**

Como sabemos se um número é múltiplo de 4?

Ora, é simples: quando o resto da divisão dele por 4 dá 0! Ou seja, quando a divisão dele por 4 é exata.

Incluimos, portanto, a cláusula if na nossa Comprehension:

```
lista_nova = [x for x in range(1,30) if x % 4 == 0]
```

Podemos traduzir como **"crie um item x, para cada x dentro da range(1,30) apenas se o resto desse x pela divisão por 4 for igual a 0 [zero]"**.

O Python irá iterar de 1 a 29 [o 30 não é incluído, lembra?] e testará se aquele número daquela iteração é divisível por 4. Se for, o número x daquela iteração é incluído na lista_nova; se o número x daquela iteração não for divisível por 4, o número não será incluído na lista_nova.

O resultado deste comando é a lista_nova ser construída com esses itens:

```
[4, 8, 12, 16, 20, 24, 28]
```

24.1.2. List Comprehensions com Texto

Não fique achando que a gente só consegue usar expressões lógicas para criar listas com números, não! O bagulho é sinistro! Dá para fazer muita coisa além disso – inclusive com texto!

Vamos ver um exemplo! Imagine a seguinte lista:

```
alunos = ["João", "Ana", "Alberto", "Pedro", "Antonio", "Mateus", "Davi",  
"Miguel", "Alice", "Bernardo", "Heitor", "Américo"]
```

E é nosso desejo criar uma segunda lista contendo apenas os nomes dos alunos que começam com a letra "A". Como fazer? Com uma expressão:

```
lista_a = [nome for nome in alunos if nome[0] == "A"]
```

A expressão acima pode ser traduzida como: **crie a lista_a contendo nome, analisado da lista alunos, nome a nome, desde que a posição 0 da string nome seja igual a "A".**

Ô João, é por que o nome[0] == "A"?

Porque uma string pode ser acessada como uma lista. Cada letra de uma string pode ser acessada separadamente, como se fosse uma "posição" na string. A primeira letra de uma string sempre é a **posição 0 [zero]** dessa string. [a segunda letra é a posição [1], a terceira letra é a posição [2], a quarta

letra da string é a posição [3] e assim por diante. O resultado desta expressão criará a **lista_a** com o seguinte conteúdo:

```
['Ana', 'Alberto', 'Antonio', 'Alice', 'Américo']
```

24.2. Vamos Exercitar List Comprehensions?

24.2.1. Descreva a lista que será criada ao escrever cada uma dessas expressões:

```
lista = [w*3 for w in range(3,20,3)]
```

```
lista = [k-1 for k in range(5,10) if k != 7]
```

```
lista = [num//3 for num in [10, 15, 19, 22, 23]]
```

```
lista = [x+4 for x in range(1,6) if x*2 > 5]
```

24.2.2. Analise o código abaixo. Qual a saída na tela?

```
lista = ['a', 'b', 'c', 'd', 'e', 'f']
nova_lista = [letra * 3 for letra in lista if letra != 'c']
nova_lista.pop()
nova_lista.reverse()
for i in nova_lista:
    print(i*2)
print("Fim do Programa")
```

Lição 25. Mais Segredos com while

25.1. Voltando ao Loop Infinito

Quando fomos apresentados aos dois laços de repetição da linguagem Python, o **for** [que usamos quando sabemos quantas vezes o laço deverá ser realizado] e o **while** [que usamos quando não sabemos], vimos que no segundo caso [no **while**], existe a possibilidade de loop infinito.

Um loop infinito acontece quando uma condição inicialmente verdadeira é testada no while e, dentro dele, nenhuma alteração nesta condição é feita. Ou seja, o while irá sempre repetir as mesmas coisas, o tempo todo... Um código de exemplo pode ser visto a seguir:

```
senha = 10
while senha > 3:
    print("repetido") # Esta linha será repetida para sempre
```

Para evitar que essa linha se repita para sempre, poderíamos adicionar uma linha que alterasse o valor da variável senha para -1 a cada loop, o que faria com que ela, em algum momento, a variável senha fique com valor 3, o que tornaria a condição do while falsa, fazendo com que o loop não continue. A solução seria essa:

```
senha = 10
while senha > 3:
    print("repetido")
    senha -= 1 # Esta linha diminui o valor de senha em 1. Garantia de
               que o loop vai parar algum dia.
```

Porém, em alguns casos, é necessário que o loop seja, realmente, "infinito" – porque não sabemos quantas vezes o loop será realizado [depende do usuário do nosso programa]. Um caso muito comum é um "menu principal" em um programa. Um menu que apresente várias opções ao usuário e que deva ser sempre mostrado, várias e várias vezes, porque o usuário o utilizará diversas vezes seguidas.

Neste caso, deve haver uma forma de dizer ao programa que o usuário não quer mais que aquele menu apareça para ele e, com isso, o programa termine. Isso é muito fácil: basta que o usuário digite que quer sair! Ou seja, basta que a condição do **while** seja alterada por meio de **algo que o usuário digite**.

Vamos ver esse exemplo? Vamos começar criando uma variável chamada **opcao** e dar a ela o valor 1, depois fazer o **while** fazer loop enquanto a variável **opcao** for diferente de 0 [zero]. O começo do nosso código é esse aqui:

```
opcao = 1
while opcao != 0:
    # Comandos dentro do Loop While
```

Isso constitui um loop infinito, percebeu? A variável `opcao` é 1 e a condição do `while` é verdadeira, caso a variável `opcao` não seja alterada, os comandos dentro do `while` vão se repetir infinitamente.

Ok... beleza... vamos agora colocar o programa mais completo, com opções de escolha pro usuário:

```
opcao = 1 # 0 Programa começa aqui, definindo opcao como 1 para o loop
executar.
while opcao != 0:
    print("=" * 30)
    print("Cadastro de Alunos")
    print("=" * 30)
    print("Incluir Aluno [1]")
    print("Alterar Aluno [2]")
    print("Excluir Aluno [3]")
    print("Sair [0]")
    opcao = int(input("Digite a Operação Desejada: ")) # Aqui, o usuário
    digita o número da opcao
# Aqui, o programa continua com as operações que você quiser...
# Quando o programa terminar, e tentar fazer outro loop desses, se a opcao
    estiver com 0, ele para!
```

Perceba que o programa começou com a atribuição do valor 1 para a variável `opcao`. Depois, imediatamente, o laço `while` começou testando se a variável `opcao` era diferente de 0 [zero] – e ela é, por isso o laço será realizado.

O programa rodando exibe algo como isso aqui:

```
=====
Cadastro de Alunos
=====
Incluir Aluno [1]
Alterar Aluno [2]
Excluir Aluno [3]
Sair [0]
Digite a Operação Desejada:
```

Nesta última linha, o usuário deverá digitar um valor que, segundo o programa, será armazenado na variável `opcao`, depois de convertido para inteiro [note lá no código o `int(input...)`].

Na parte inferior do código, após o menu principal, coloquei apenas dois comentários, mas o lógico a fazer é tratar o que o usuário digitou e fazer o programa rodar a operação que o usuário quer executar [Incluir, Alterar ou Excluir alunos, ou, por fim, sair do programa].

Vamos fazer o programa certo?

Pule para a próxima página e veja os dois comandos que quero apresentar nesta lição:

```
opcao = 1
while opcao != 0:
    print("=" * 30)
    print("Cadastro de Alunos")
    print("=" * 30)
    print("Incluir Aluno [1]")
    print("Alterar Aluno [2]")
    print("Excluir Aluno [3]")
    print("Sair [0]")
    opcao = int(input("Digite a Operação Desejada: "))
    if opcao == 1:
        #Faça os comandos para Incluir Aluno
    elif opcao == 2:
        #Escreva aqui os comandos para Alterar o cadastro do aluno
    elif opcao == 3:
        # Escreva aqui os comandos para Excluir um aluno.
    elif opcao == 0:
        break
    else:
        continue
# Fim do Programa (já estamos fora do loop while)
```

Note que dependendo o valor da variável `opcao`, o programa fará coisas diferentes [o **if**, seguido de vários **elif** e de um **else** fazem isso]... Neste "catatau" de coisa, eu só queria mostrar especificamente dois comandos: o comando **break** e o comando **continue**.

O comando **break**, no código acima, está associado à cláusula **elif opcao==0**, ou seja, se o usuário digitar 0, o comando que será executado é justamente esse **break**. E a função dele é bem simples: ele encerra o loop. Ele faz o Python sair do loop. Quando o Python lê o comando **break**, ele sai do **while** e começa executando o primeiro comando fora do loop.

Em suma, este comando é o que se usa para **sair de um loop** [seja ele infinito ou não].

Note bem: assim que o Python lê o comando **break**, ele salta pra fora do loop. Ele deixa de executar os comandos posteriores dentro do loop. Simplesmente ignora eles.

O comando **continue** tem uma "missão" diferente. Quando o Python lê este comando, ele salta para o **início do loop**, ou seja, ele ignora tudo o que vem depois e volta ao início do loop para reiniciar todos os

comandos do loop. No caso de um loop **for**, ele **itera o for**, ou seja, ele salta para o início do loop e conta o próximo valor do for.

São dois comandos que ajudam muito no fluxo de um programa, essencialmente no **while** e no **for**.

25.2. Vamos Exercitar?

25.2.1. Qual a saída para o código a seguir?

```
# print("Programa de Exemplo")
print("=" * 30)
senha = 'x'
while senha != 'z' :
    if senha == 'a' :
        print("primeira passagem")
        break
        senha = "b"
    elif senha == 'x' :
        print("segunda passagem")
        senha = 'c'
        continue
        senha = 'x'
    elif senha == 'c' :
        print("terceira passagem")
        senha = 'a'
        #continue
    else:
        print("última passagem")
        break
    print("-" * 30)
print("-" * 30)
print("fim do programa")
```

Lição 26. Brincando mais com Listas e Strings

26.1. Um pouco mais sobre índices

Strings e **Lists** são diferentes entre si e nós sabemos disso! **String é um tipo de dado** [um formato de aceitação de dado] que pode ser manipulado pelo Python e **List é uma estrutura de dados** [um objeto capaz de receber e armazenar dados diversos] que o Python entende.

Apesar de serem essencialmente diferentes, há muita coisa que podemos fazer em comum [e da mesma maneira] em ambos. O uso de índices é uma delas:

```
a = "Mises"
b = ['M', 'i', 's', 'e', 's']
print(a[0])
print(b[0])
```


No código acima, estabelecemos que a **variável a** vai receber a **palavra "Mises"** e que a **variável b** vai receber **a lista composta pelas 5 letras da palavra "Mises"**. Os conteúdos são bem diferentes entre si, mas pedir para mostrar na tela **a[0]**, que significa apontar para a primeira letra da string "Mises" [portanto, para a letra "M"] é o mesmo que apontar para **b[0]**, que faz referência ao primeiro item da lista b, ou seja, a letra "M" também.

Pra resumir, e nós já sabemos disso, cada letra da string pode ser acessada diretamente pelo seu índice, do mesmo jeito que numa lista.

26.1.1. Índice Negativo

É possível usar, tanto em listas quanto em strings, uma referência a um índice negativo, como no exemplo **nome[-2]**. O que isso significa?

Bom, sabemos que listas e strings começam com índice 0 [zero], que representa o primeiro item da lista ou o primeiro caractere de uma string. Veja a seguir um exemplo de uma variável chamada **estado**:




P	E	R	N	A	M	B	U	C	O
0	1	2	3	4	5	6	7	8	9

Note que o item **estado[4]** aponta para a letra **"A"**. Por sua vez, o item **estado[8]** aponta para **"C"**.

Se usarmos índices negativos, a contagem começa no **-1**, a partir **do último caractere ou item da lista**, contando para trás, como se pode ver a seguir:

P	E	R	N	A	M	B	U	C	O
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



Portanto, o item **estado[-4]** aponta para a letra **"B"** e o item **estado[-8]** aponta para **"R"**.

26.2. Slices [Fatias]

Quando nós usamos variável[índice], estamos apontando para um único objeto, ou seja, estamos obtendo um único resultado como resposta – seja um caractere, vindo de uma string, ou um único item, vindo de uma lista.

Podemos, porém, fazer uso de slices [a gente fala "isláicis", e quer dizer "fatias"] para obter mais de um caractere ou mais de um item de uma lista. É super fácil de usar! Veja o código a seguir:

```
a = [1, 4, 9, 16, 25, 36, 49, 64]
b = a[2:5]
print(b)
```

Note que, em primeiro lugar, criamos uma lista a contendo 8 itens. Na segunda linha, pedimos para criar uma lista b contendo uma "fatia" da lista a, que vai **do item 2 até o item 5** [o sinal de "dois-pontos" é usado como "até"]. Ao pedir para printar a lista b, o resultado será:

```
[9, 16, 25]
```

Peraí, João... se falamos "de 2 até 5", isso tem que incluir, na listagem acima, os índices 2, 3, 4 e 5, ou seja, pegar o número 36 também!

Bem pensado, amigo leitor, querida leitora! Mas aí é que tá... quando apontamos para [2:5], o **índice final não é incluído na fatia** – lembrem-se da função **range()**? Do mesmo jeito que ela... também nas fatias, o último item não é incluído.

Então, se fizermos **estado[3:7]** no exemplo que usamos na página anterior, estamos apontando para:

P	E	R	N	A	M	B	U	C	O
0	1	2	3	4	5	6	7	8	9

Note que ele inclui a posição 3, mas não inclui a 7, indo até a 6, apenas. Veja isso no Shell:

```
>>> a = "Pernambuco"
>>> b = a[3:7]
>>> print(b)
namb
```

Podemos pedir fatias **"do 3 até o final"** ou **"do início até o 4"** ou **"do penúltimo até o final"**, usando alguns códigos especiais antes e depois do dois-pontos, vejamos:

```
a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
b = a[:5] # do início até o índice 4 [não inclui o 5]
c = a[6:] # do 6º item até o final
d = a[-3:] # do ante-antepenúltimo item até o final
e = a[:-4] # do início ao 4º - item contando a partir do final
```

Se quiser começar a slice **a partir do início** da lista ou string, simplesmente **não coloque nada antes do sinal de dois-pontos**. Se quiser que a sua fatia **vá até o final** da lista ou string, **não coloque índices após o sinal de dois-pontos**.

Ahhh, a propósito, as respostas para as variáveis mostradas no exemplo anterior são:

```
>>> print(b)
[1, 4, 9, 16, 25]
>>> print(c)
[49, 64, 81, 100]
>>> print(d)
[64, 81, 100]
>>> print(e)
[1, 4, 9, 16, 25, 36]
```

26.2.1. Comando del em Fatias

Você pode usar o comando del [que apaga variáveis] para apagar diversas posições de uma lista, basta que aponte para o slice [fatia] desejado. Veja o exemplo abaixo:

```
>>> lista = [2, 4, 6, 8, 10, 12, 14, 16]
>>> del lista[2:5]
>>> print(lista)
[2, 4, 12, 14, 16]
```

Verifique que o comando **del lista[2:5]** só apagou os itens nas **posições 2, 3 e 4** da lista. Porque, você já sabe, **a fatia 2:5 não inclui o item na posição 5** [nunca inclui o item na posição depois dos dois-pontos].

26.3. Listas são Mutáveis, Strings são Imutáveis

Uma diferença muito importante entre listas e strings é que no caso das listas, **é possível mudar um item específico** [trocar o seu valor] acessando a posição certa por meio do índice. **Não é possível, porém, fazer o mesmo em relação a uma string**. Não dá para alterar apenas um caractere específico numa string – por isso dizemos que strings são imutáveis e listas são mutáveis [podem mudar].

```
texto = "Mises"
lista = ["M", "i", "s", "e", "s"]
lista[2] = "c"      # Isso coloca "c" na posição 2 da lista
texto[2] = "c"      # Isso vai dar um erro – é impossível fazer isso
```

Dá pra fazer, normalmente, a alteração do item 2 da lista lista. Mas não é possível alterar o caractere 2 da string texto. Só dá para alterar o valor inteiro da string, nunca um caractere específico.

26.4. Vamos Exercitar?

26.4.1. Analise o Código a seguir:

```
texto = "Pernambuco"
lista = [1, 10, 100, 1000, 10000, 100000]
fat1 = texto[:5]
fat2 = lista[-3:]
fat3 = texto[-6:-2]
fat4 = lista[1]
lista[0] = 20
texto[0] = "T"
```

26.4.2. Como resolver o problema no código acima digitando apenas um único caractere?

26.4.3. O que será impresso pelo comando `print(fat1 * fat4)`?

26.4.4. O que será impresso pelo comando `print(fat3 * (lista[0]/10))`?

26.4.5. O que será impresso pelo comando `print(fat1[3:]*(fat4/2))`?

Lição 27. Tuplas

27.1. Sobre o que já sabemos...

A última lição nos mostrou as semelhanças entre as listas [estrutura de dados] e as strings [tipo de dados]: ambas admitem ser acessadas por meio de índices na forma de **nome[índice]** e também permitem o uso de slices [fatias] no formato **nome[início:final]**.

Lembrando que **listas são mutáveis** [ou seja, podemos mudar um item específico nelas] e **strings são imutáveis** [ou seja, não podemos alterar um item específico nelas por meio do acesso ao índice].

Chegou a hora de aprender sobre uma outra estrutura de dados: a **tupla** [tuple em inglês].

27.2. Conhecendo as Tuplas

Tupla é o nome dado a uma **estrutura de dados parecida com a lista**, mas que é **imutável** [ou seja, não dá para modificar isoladamente um item de uma tupla].

Para construir uma tupla, você pode atribuir a uma variável diversos valores separados por sílabas. Você pode colocar os valores entre **()** [parênteses] ou não [isso é opcional].

```
nomes = 'Mises', 'Hayek', 'Rothbard', 'Beltrão', 'Iorio'  
idades = (103, 98, 95, 51, 63)
```

Nas duas linhas mostradas acima, temos duas formas de criar tuplas: com e sem parênteses. O Python aceita as duas formas de escrita!

Poxa, João, se uma Tupla é imutável, depois de criada não dá para guardar novas coisas nela, né? Para que ela serve, então?

Excelente pergunta! Uma tupla é usada quando precisamos de uma estrutura "somente-leitura", ou seja, de um espaço na memória onde **só é possível consultar** os dados já guardados, e **não modificá-los** – acredite em mim, amigo leitor, amiga leitora: vão acontecer casos em que é necessário, sim, usar estruturas para "somente leitura".

Uma tupla não pode ter itens apagados, nem adicionados, nem modificados... Se ela nasceu com 5 itens, ela vai existir sempre com aqueles 5 itens! Nunca serão apagados, nunca serão modificados, não será possível adicionar novos itens na tupla.

27.3. Como Criar uma Tupla vazia?

Apesar de ser algo aparentemente inútil, é possível criar uma Tupla vazia, basta escrever:

```
t1 = ()
```

Também é possível criar uma tupla com um único item dentro dela, mas para isso é necessária atenção especial de sua parte, amigo leitor! Para criar uma tupla com um único item, **você deve escrever o item e uma vírgula depois dele**, caso contrário o Python entenderá que você quer criar uma variável simples, olha só:

```
t1 = (1,)      # Isso é uma tupla com apenas o elemento 1
t2 = 2,        # Isso é uma tupla com apenas o elemento 2
t3 = (1)       # Isso é uma variável contendo um inteiro 1, apenas.
t4 = 2         # Isso é uma variável contendo apenas o inteiro 2
```

Depois de criar uma tupla, como você bem sabe, não é possível alterar nenhum de seus itens.

27.4. O que podemos Fazer com Tuplas, então?

As operações básicas que podemos realizar com Tuplas são:

27.4.1. Exibir o tamanho da Tupla

Podemos exibir o tamanho [comprimento] da Tupla usando a função **len()**, conforme vemos no exemplo a seguir:

```
>>> tupla1 = (2, 4, 6, 8, 10, 12, 14, 16)
>>> print(len(tupla1))
8
```

A tupla1 tem 8 elementos e usar a função len() nela resultará no número 8.

27.4.2. Concatenar Tuplas

É possível unir duas ou mais tuplas em uma única [uma nova tupla]. Para isso, basta usar o **operador + [sinal de mais]**. Veja o exemplo a seguir:

```
>>> t1 = 1, 2, 3
>>> t2 = 4, 5, 6
>>> t3 = t1 + t2
>>> print(t3)
(1, 2, 3, 4, 5, 6)
```

Note: não houve alteração em nenhuma tupla [já que não é possível alterá-las]. Houve apenas a criação de uma terceira tupla chamada t3 a partir das duas anteriores [t1 e t2].

27.4.3. Repetir Tuplas

Uma tupla pode ser repetida por meio do **operador * [asterisco]**, exatamente como fazendo com strings. Veja o exemplo a seguir:

```
>>> t1 = 1, 2, 3
>>> t2 = t1 * 3
>>> print(t2)
```

```
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Note que os itens da tupla t1 foram multiplicados por 3 para criar a tupla t2.

27.4.4. Teste de Pertencer

É possível testar se um item pertence, ou não, a uma tupla, usando o **operador in** ["em" em português], exatamente como usamos em listas. Veja a seguir:

```
>>> t1 = 1, 2, 3, 4, 5
>>> print(4 in t1)
True
>>> print("4" in t1)
False
```

Ao pedir para imprimir **4 in t1**, obtemos o resultado **True**, porque o 4 existe em t1. No momento em que pedimos para mostrar **"4" in t1**, a resposta é **False** porque não existe "4" [string] dentro de t1.

27.4.5. Iteração [para o laço for]

Uma tupla pode ser usada como objeto iterável num laço for, exatamente como uma lista. Veja o exemplo a seguir:

```
t1 = 1, 2, 3, 4, 5, 6
for i in t1:
    print(i**2)
```

O que este programa faz é varrer a tupla t1, item a item [iteração], e imprimir na tela o valor do número em questão elevado ao quadrado [elevado à potência 2]. O resultado do programa será:

```
1
4
9
16
25
36
```

27.5. Vamos Exercitar sobre Tuplas

27.5.1. Qual o resultado do programa abaixo mostrado na tela?

```
t1 = 2, 4, 6, 8
t2 = (1, 3, 5, 7, 9)
t3 = t1 + 2 * t2
print(len(t3)*t1[1])
print(t2[1]*2 in t3)
```

Lição 28. Semelhanças entre Tuplas e Listas

28.1. O Porquê desta Lição

Como Tuplas e Listas são muito parecidas, resolvi trazer algumas comparações entre essas duas estruturas de dados, mostrando, inicialmente, as principais...

28.2. Semelhanças entre Tuplas e Listas

28.2.1. Criando Listas e Tuplas

Listas e Tuplas são objetos com vários conteúdos [chamados de "estruturas de dados"] e, por isso, admitem receber vários itens. Para identificar esses vários itens, separamos eles por vírgulas.

Tuplas podem ser criadas com os **itens separados por vírgulas diretamente** ou por eles envolvidos por **parênteses**. **Listas** são criadas com vários itens separados por vírgulas envolvidos por **colchetes**.

```
e1 = 1, 2, 3, 4      # Isso é uma TUPLA
e2 = (1, 2, 3, 4)    # Isso também é uma TUPLA
e3 = [1, 2, 3, 4]    # Isso é uma LISTA
```

Para criar listas e tuplas vazias, basta criar a variável atribuindo colchetes ou parênteses vazios:

```
e4 = ()              # Isso é uma TUPLA vazia
e5 = []              # Isso é uma LISTA vazia
```

Para criar uma lista com apenas um elemento, atribua à variável o valor entre colchetes [apenas isso].

Para criar uma tupla com um único elemento, deve-se atribuí-lo seguido da vírgula para evitar que o Python entenda errado [como vimos na lição anterior]:

```
e6 = (1,)            # Isso é uma TUPLA contendo apenas o inteiro 1
e6 = 1,              # Isso é uma TUPLA contendo apenas o inteiro 1
e7 = [1]             # Isso é uma LISTA contendo apenas o inteiro 1
```

Não esqueça: Listas e Tuplas podem conter **diversos tipos de dados diferentes**: Strings, Inteiros, Floats, Booleanos etc, conforme o exemplo a seguir:

```
lista1 = [1, 2, "a", 3, "c", True, 4, 8.7]
tupla2 = (1, 2, "c", 4, False, True, 5, "20", 14.9)
```

28.2.2. Apagando Listas e Tuplas

O comando **del** pode ser usado tanto em Tuplas quanto em Listas [e também é usado em variáveis comuns do mesmo jeito]. Este comando apaga o objeto em questão retirando-o definitivamente da memória do computador. Ou seja, o objeto apagado pelo **del** deixa de existir.

```
t1 = 1, 2, 3, 4
l1 = [1, 2, 3, 4]
```

```
v1 = "texto"
del t1, l1, v1          # Este comando apaga as três variáveis
```

28.2.3. Acessando Posições das Listas e Tuplas

Tanto em tuplas quanto em listas, é possível acessar uma posição específica da estrutura por meio do uso do índice. Usa-se o nome da variável seguido do número da posição **entre colchetes**.

```
>>> t1 = "Mises", "Hayek", "Hoppe", "Menger", "Bohm-Bawerk"
>>> l1 = ["Mises", "Hayek", "Hoppe", "Menger", "Bohm-Bawerk"]
>>> print(t1[3])
"Menger"
>>> print(l1[0])
"Mises"
```

Também é possível acessar os itens de uma lista ou tupla por meio de índices negativos, que começam do -1, que é o último item da lista/tupla [sempre contando do último item para o primeiro].

```
>>> print(t1[-3])
"Hoppe"
>>> print(l1[-1])
"Bohm-Bawerk"
```

Também é possível fazer **slices [fatias]** apontando para alguns itens usando o **operador :** [dois-pontos] tanto em listas como em tuplas.

```
>>> print(t1[1:4])
('Hayek', 'Hoppe', 'Menger')
>>> print(l1[-3:-1])
['Hoppe', 'Menger']
```

28.2.4. Concatenando Tuplas e Listas

É possível concatenar Tuplas e Listas, basta usar o **operador +** [sinal de mais]. O resultado é uma lista/tupla com a soma dos elementos das duas ou mais originais:

```
>>> l1 = [1,2,3,4]
>>> l2 = [4,5,6,7]
>>> l3 = l2 + l1
>>> print(l3)
[4, 5, 6, 7, 1, 2, 3, 4]
```

Não dá para concatenar uma tupla com uma lista. Só é possível concatenar estruturas de dados de mesmo tipo [tupla com tupla, lista com lista].

28.2.5. Teste de Pertencimento

É possível usar o **operador in** para testar se um determinado item está numa lista ou tupla.

```
>>> l1 = [1, 2, 3, 4, 5, 6]
```

```
>>> t1 = 1, 2, 3, 4, 5, 6
>>> print(2 in t1)
True
>>> print(4 in t1)
True
```

28.2.6. Usando Tuplas e Listas como objetos iteráveis no laço FOR

Como já estamos cansados de saber, o laço for é usado para repetir várias vezes um determinado conjunto de operações desde que se apresente um objeto iterável como "mapa" pelo qual o laço for tem que "passar", item a item.

Esse objeto iterável pode ser uma função **range()**, uma **lista** ou uma **tupla** [entre outras opções]. Segue um exemplo com uma tupla, mas poderíamos apresentar t1 como sendo uma lista ou uma variável construída com range():

```
t1 = 1, 2, 3, 4, 5, 6
for i in t1:
    print("o triplo de", i, "é", i*3)
```

O resultado deste programa é:

```
o triplo de 1 é 3
o triplo de 2 é 6
o triplo de 3 é 9
o triplo de 4 é 12
o triplo de 5 é 15
o triplo de 6 é 18
```

28.2.7. Funções usadas em Listas e Tuplas

Há ainda algumas funções que podemos usar tanto em listas quanto em tuplas. Vamos listá-las agora para você, amigo leitor, amiga leitora:

Função	Descrição
len(objeto)	Retorna o tamanho [comprimento] do objeto [tupla ou lista]. Ou seja, retorna o número de itens do objeto.
max(objeto) e min(objeto)	Retornam o maior [max] e o menor [min] valor dentro do objeto [tupla ou lista]
tuple(sequência)	Converte uma sequência [lista ou range()] em tupla .
list(sequência)	Converte uma sequência [tupla ou range()] em lista .

28.3. Vamos Exercitar?

28.3.1. Qual o resultado na tela dos códigos a seguir?

```
lista = []
lista.append(1)
tupla = 1, 2, 3, 4, 5, 6, 7
for i in tupla:
    lista.append(i)
tupla = list(tupla)
print(lista == tupla)
```

```
tupla = (3)
variavel = 3
lista = [1, 2, 3]
lista = lista * variavel
tupla = tupla * variavel
print(lista)
print(tupla)
```

```
tupla = [8, 0, 9, 3]
variavel = 2
lista = (1, 2, 3)
lista = list(lista)
tupla = tuple(tupla)
tupla = variavel * lista
print(tupla + lista)
```


Lição 29. Dictionaries [Dicionários]

29.1. Pequena Apresentação

Um **Dictionary [Dicionário]** é uma estrutura de dados como uma lista ou uma tupla. Dicionários são, porém, estruturas de dados mais complexas que Listas e Tuplas. Um dicionário é formado por vários elementos, e cada um desses elementos é formado por uma **dupla de informações** que chamamos de **Chave e Valor**.

Dicionários armazenam dados dentro dessas duplas. Os dados armazenados são os valores e a forma de encontrar esses valores são justamente as chaves. O acesso aos dados não é feito por meio de índices, mas por meio das chaves.

29.2. Criando Dicionários

Para construir um dicionário, você escreve assim:

```
| dicio = {'Nome': 'João', 'Idade': 42, 'E-mail': 'joaonovo30@yahoo.com'}
```

Note que o Dicionário é envolvido por **chaves** [o nome desses símbolos: **{}**] e cada elemento da listagem é composto de **chave : valor** [separados pelo **sinal de :** - dois-pontos] e os elementos são separados, entre si, por vírgulas, como nas listas e tuplas. Os valores de um dicionário são mutáveis, podem ser alterados livremente, mas as chaves que apontam para eles têm que ser imutáveis, como uma string, um número ou uma tupla.

Para acessar um determinado elemento num dicionário, é necessário apontar para o nome do dicionário seguido do nome da chave [entre colchetes] do elemento desejado. Assim:

```
| print(dicio['Nome'])
```

Perceba que este código aponta para a chave **'Nome'** dentro do dicionário **dicio**. O resultado deste código é bem simples: ele retorna o valor associado à chave 'Nome', ou seja:

```
| 'João'
```

Um dicionário é, portanto, uma estrutura de dados que se assemelha a algo assim:

Nome	Idade	E-mail
João	42	joaonovo30@yahoo.com

Para criar um dicionário vazio, é suficiente atribuir **{}** [abrir e fechar chaves] ao nome do dicionário que se deseja criar. Acho que você já tinha deduzido, né? É muito semelhante a listas e tuplas:

```
| dicionario_vazio = {}
```

29.3. Alterando Valores dos Itens num Dicionário

Para alterar Valores em um dicionário, basta atribuir um valor novo a uma chave já existente. Ou seja, para mudar o valor da chave `Idade`, é só fazer isso:

```
dicio['Idade'] = 43  
print(dicio)
```

O resultado disso é:

```
{'Nome': 'João', 'Idade': 43, 'E-mail': 'joaonovo30@yahoo.com'}
```

Perceba que o valor associado à chave `'Idade'` foi alterado.

29.4. Criando um Novo Elemento no Dicionário

Para criar itens novos, basta atribuir valores a chaves que ainda não existem – essas chaves serão criadas e preenchidas com o valor que você está inserindo agora. Por exemplo, olha o código a seguir:

```
dicio['Telefone'] = '98213-2325'  
print(dicio)
```

O resultado disso é:

```
{'Nome': 'João', 'Idade': 43, 'E-mail': 'joaonovo30@yahoo.com',  
'Telefone': '98213-2325'}
```

Ou seja, nosso dicionário passou a ser semelhante a esta tabela:

Nome	Idade	E-mail	Telefone
João	42	Joaonovo30@yahoo.com	98213-2325

29.5. Apagando Elementos do Dicionário

Para apagar um elemento de um dicionário, use o comando **del**. Veja um exemplo:

```
del dicio['E-mail']      # Aqui o elemento da chave 'E-mail' foi apagado  
print(dicio)
```

O resultado deste código é:

```
{'Nome': 'João', 'Idade': 43, 'Telefone': '98213-2325'}
```

Note que o elemento que possuía a chave `'E-mail'` foi excluído do dicionário. Para apagar o dicionário inteiro [não somente um elemento específico], é só usar **del dicio** [isso destrói o dicionário inteiro].

Também é possível "esvaziar" um dicionário [apagar todos os seus elementos, mas deixar o dicionário existindo vazio, sem itens], para isso usamos o método `.clear()`.

```
dicio.clear()           # Aqui o dicio foi esvaziado  
print(dicio)
```

Esse código resultará em:

```
{}
```

29.6. Chaves Duplicadas?

Um dicionário **não aceita mais de um elemento com a mesma chave**. Se um dicionário for criado com dois elementos com a mesma chave [mesmo nome], apenas o posterior ficará [não é o primeiro que vai ficar, é o último]. Veja esse exemplo:

```
dic = {1: "Mises", 2: "Hayek", 3: "Menger", 2: "Rothbard", 3: "Hoppe"}
```

Note que há dois elementos com a chave 2 e dois elementos com a chave 3. Isso vai resultar em:

```
>>> print(dic)
{1: "Mises", 2: "Rothbard", 3: "Hoppe"}
```

O elemento **2:"Hayek"** e o elemento **3:"Menger"** foram desconsiderados, sendo mantidos apenas os outros dois posteriores com as chaves 2 e 3.

29.7. Vamos Exercitar?

29.7.1. Qual o resultado do código abaixo?

```
lista = []
dicio = {}
seq = range(10)

for i in seq:
    lista.append(i*2)
    dicio[i] = i**2

lista.reverse()
lista.pop()
lista.remove(10)
del dicio[3]
# print(lista)
# print(dicio)
print(lista[4] * dicio[2] - 2)
```