1. No Python, toda variável é um objeto.
   1. Um objeto é um pouco mais do que uma variável.
2. nome = input("Informe seu nome: ")

#print ("Prazer em te conhecer, **{:20}**!".format(nome))

#No exemplo acima, ele vai escrever o nome num espaço de 20 caracteres

# Vai ficar: "Prazer em te conhecer, antonio             !"

#print ("Prazer em te conhecer, **{:>20}**!".format(nome))

#Acima, com ">", vai alinhar o nome à direita

# Vai ficar: "Prazer em te conhecer,                 João!"

#print ("Prazer em te conhecer, **{:<20}**!".format(nome))

#Com "<", vai alinhar à esquerda

#print ("Prazer em te conhecer, **{:^20}**!".format(nome))

#Com "^", vai alinhar ao centro

#print ("Prazer em te conhecer, **{:=^20}**!".format(nome))

# No modelo acima, vai alinhar ao centro e imprimir "=" nos espaços vazios

# Vai ficar: "Prazer em te conhecer, ========João========!"

# MODELO MAIS NOVO:

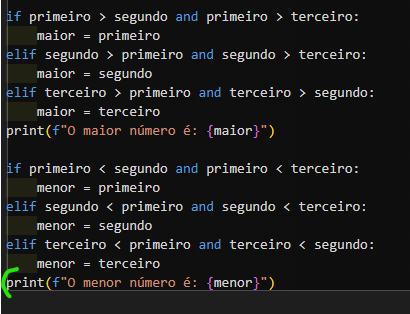
print(f'Prazer em te conhecer, **{nome:=^20}**!'.upper())

1. **O Pyhton aceita variáveis com acentos!**
2. Só usar variável quando esse valor precisar ser resgatado futuramente. Se for só para fins matemáticos, fazer a conta direto no programa para economizar memória.
   1. Pode parecer pouco em programas simples, mas a memória poupada faz diferença em programas maiores e mais complexos.
3. Para importar uma biblioteca, basta digitar “import” e o nome da biblioteca.
   1. Ex: **import math**.
   2. Também podemos importar só um elemento da biblioteca, como: **from math import sqr**t.
   3. Para consultar a biblioteca do Python: <https://docs.python.org/pt-br/3/library/index.html>
   4. Para baixar bibliotecas extras: <https://pypi.org/>
4. Para usar emojis no Python, pegar os códigos de: <https://apps.timwhitlock.info/emoji/tables/unicode>
   1. Substituir o “+” do código unicode por “000” e colocar uma “\” antes de tudo. Então, “U+1F601” deve ficar “\U0001F601”.

# STRING

1. Cadeia de texto é qualquer string, que é reconhecida pelo Python como texto.
   1. Ex: **frase = “curso de Python”**.
   2. Cada caractere da string, incluindo os espaços, ocupa um micro-espaço na memória do computador.
   3. Ex: 
   4. Com essa característica é possível fatiar a string e manipulá-la. Ex: 
      1. Ao imprimir “**frase[9]**”, o programa vai mostrar o caractere de número 9 dessa string. No caso, será a letra “V”, conforme a divisão mais acima.
   5. 
      1. Ao escrever **frase[9:13]**, o programa vai pegar do 9 até o 13 e excluir o 13. Ou seja, vai imprimir só 9, 10, 11 e 12. É sempre um a menos no final. O começo é certinho.
   6. 
      1. Por outro lado, para conseguir pegar do 9 ao 20, devemos pedir para o programa imprimir do 9 ao 21; embora nossa string vá apenas até 20.
   7. 
      1. No exemplo acima, a instrução significa que o programa vai pegar do 9 ao 21 pulando de dois em dois. Ou seja, neste caso só os espaços ímpares serão considerados.
   8. 
      1. Quando não especificamos o início, o programa começa de 0. No exemplo acima, então, **frase[:5]** é o mesmo que escrever frase**[0:5]**.
   9. 
      1. Da mesma forma, ao não indicar o final, o programa segue até o último elemento. Então, **frase[15:]** seria o mesmo que **frase[15:21]**. Porém, é melhor usar essa forma com o final não especificado do que colocar até um número acima do limite.
   10. 
       1. Seguindo a mesma lógica, o exemplo acima vai do elemento 9 até o final, mostrando de 3 em 3.
   11. Se colocar frase [::2], ele vai pegar do primeiro ao último elemento de 2 em 2.
   12. Também podemos atribuir isso às listas. Ex de lista: [‘João’, ‘Carlos’, ‘Roberto’].
       1. print[0] vai mostrar o primeiro item da lista, que é ‘João’.
       2. Print[0][2] vai mostrar o terceiro caractere do primeiro item da lista, que é “ã”.
2. Para imprimir textos grandes com quebra de linha, basta usar o **print** e “””(três aspas duplas) no início e mais “”” onde deve ser o final.

# Estruturas condicionais:

1. Simples: só com um if (pode acontecer ou não).
2. Composta: mais de um if e/ou um else (acontece uma coisa ou outra)
3. O elemento que estiver deslocado para a direita, abaixo da condição, será executado se a condição for cumprida. O que estiver alinhado à esquerda será executado independentemente disso. Ex:
   1. 
4. While e for: são o que chamamos de laços/repetições/iterações.
5. No python, dizemos que as variáveis são **declaradas**, não inicializadas.

# Variáveis compostas:

1. **Tuplas**
   1. Imutáveis. É preciso parar o programa, mudar a tupla e executar de novo.
   2. Não é possível atribuir valores fora da tupla.
   3. Pode usar () ou nada nas extremidades.
   4. É possível estabelecer uma tupla **a** e uma tupla **b** e depois somar **a** + **b** em uma tupla **c**. a + b é diferente de b + a, pois muda a ordem dos elementos. Ele vai juntar as duas listas numa só.
   5. Pode ter tipos diferentes de dados, como números e strings dentro da tupla.
2. **Listas**
   * 1. São mutáveis.
     2. pode ser feito com [] ou list().
     3. Cuidado na hora de igualar duas listas (**a = b**). Porque qualquer alteração feita numa delas será aplicada a ambas. a = b não cria uma cópia de uma lista na outra — não atribui os valores de uma lista à outra — mas as iguala.
     4. Porém, ao fazer “**b = a[:]**” criamos de fato uma cópia dos valores de “a” dentro de “b”, e não ligamos as duas listas.
     5. O mesmo acontece se fizer a importação de uma lista em outra. Ex: **lista2.append(lista1)**. Isso cria uma relação entre as duas listas, e, sempre que os dados da **lista1** forem alterados, os da **lista2** também serão. A alternativa, então, é criar uma cópia da **lista1** na **lista2**: **lista2.append(lista1[:])**
     6. print(\*num, sep= ', ')# '\*' mostra a lista sem os colchetes, e sep é a string entre as variáveis da lista

input:[1, 2, 3]

output: 1, 2, 3

1. **Dicionários**
   1. Pode ser feito com {} ou dict().
   2. Ao estabelecer as identificações de cada parte do dicionário, aquele ponto é reconhecido pela nomenclatura escolhida, e não mais pelo número. Ex: Se o primeiro elemento é ‘nome’, então é possível usar **print(dicionario[‘nome’])**, mas não **print(dicionario[‘0’])**, porque a posição **dicionario[‘0’]** não existe mais. Ela foi substituída pela posição **dicionario[‘nome’]**.

# Funções

1. Função = rotina
2. Exemplos de funções no Python:
   1. print()
   2. len()
   3. input()
   4. float()
   5. int()

|  |
| --- |
| “def nomeDaFuncao():  comando |

1. Defini-se com
   1. Na prática, é como o “function” do JavaScript
2. **Empacotamento**
   1. Permite empacotar e desempacotar uma série de entradas. É muito útil para definições com variáveis, como “**def soma(a, b)**”. Nessa maneira, o programa só aceita dois valores. Se quisermos mais, basta usar o empacotamento, como no exemplo:  
      
   2. Conforme o exemplo, para empacotar basta passar (\*num) ao invés de (a, b).
3. Também podemos trabalhar com listas, como neste exemplo em que criamos uma função para dobrar valores:

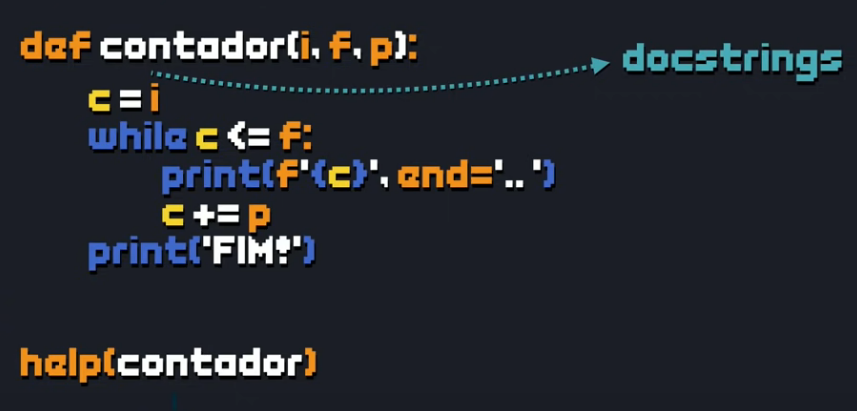


# Interactive Help

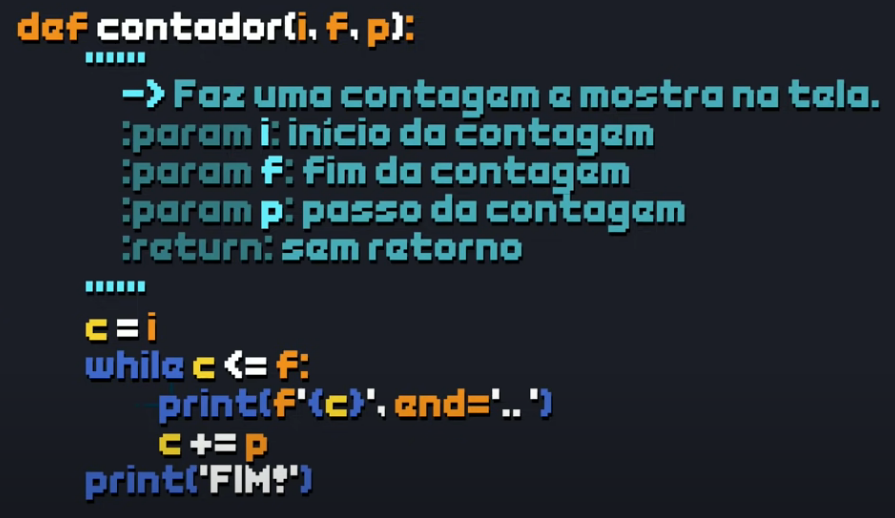
1. Para usar, basta executar a função “help()”
   1. Uma vez executado o **help()**, basta escrever o comando a ser consultado no novo prompt aberto. Para sair do **help**, basta digitar "**quit**"
2. Também é possível acessar o menu de ajuda direto pelo código.
   1. Ex: "**help(print)**" vai abrir a explicação de como funciona o comando **print**.
3. Outra maneira: “**print(input.\_\_doc\_\_)**”.

# Docstrings

1. É comum recebermos códigos e não entendermos o que significam partes dele.



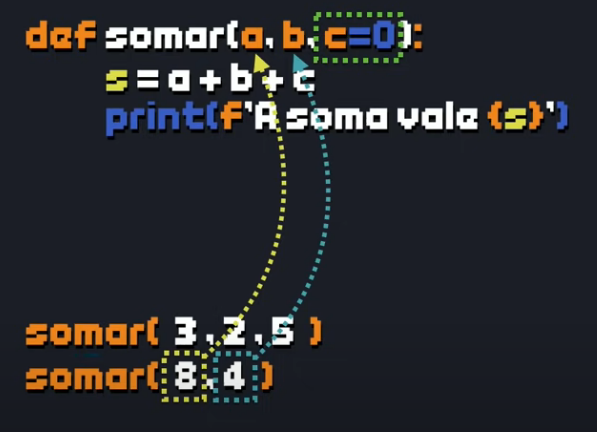
1. Para tanto, pode ser útil o criador do código usar as **docstrings** para explicar:



1. Ao digitar três aspas duplas (“) e der enter, o python já cria um mini guia (marcado mais escuro na imagem acima).
   1. É importante que a docstring seja criada na linha imediatamente abaixo à da função criada. Antes das execuções dela.
2. Depois, ao usar **help(contador)**, o python vai explicar como funciona aquela função.

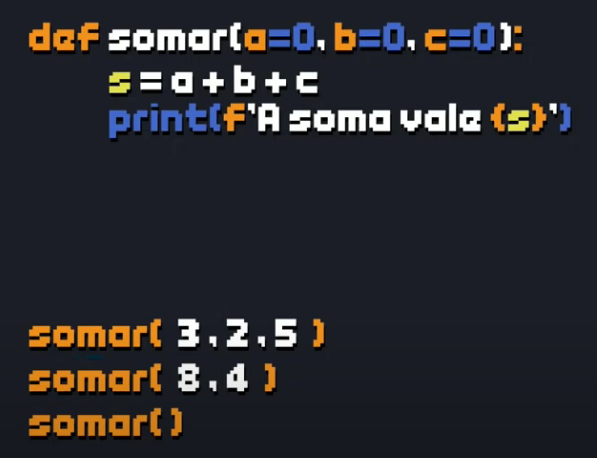
# Parâmetros Opcionais

1. Caso uma função exija um número X de parâmetros e, ao longo do código, sejam passados menos valores do que o número de parâmetros exigidos, podemos pré-estabelecer o quanto eles valerão na ausência de valor:



Dessa forma, no **somar(3, 2, 5)** tudo funcionará normalmente. Na segunda soma, com os valores 8 e 4, o **c** irá valer zero na falta de um terceiro valor para o parâmetro **c**.

1. Também é possível deixar todos os parâmetros como opcionais, o que evitaria erros na execução da função:



# Escopo de Variáveis(/Declarações)

1. Uma variável declarada fora de uma função vale em qualquer parte do programa. Porém, uma variável declarada dentro de uma função só vale nos limites da função:



1. Portanto, é possível resgatar o valor de “**a**” em qualquer parte do programa. Mas tentar usar os valores de **b** ou **c** irá dar erro.
2. Uma particularidade do Python é que, ao declarar dentro da função uma variável que existe fora dela, o programa vai ter duas variáveis; uma local e outra global:



* 1. No exemplo acima, existe um **a** local que vale 8 e um **a** global que vale 5.

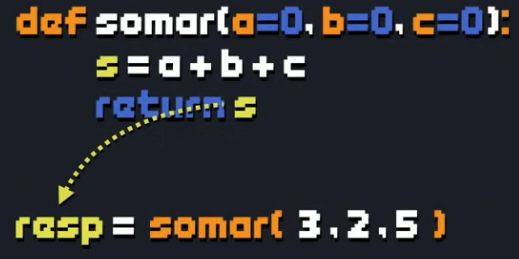
1. No entanto, podemos pedir pra função usar a variável de fora ao invés de criar outra dentro. Para isso, passamos o comando “**global [nome da variável]**”:



* 1. É importante notar que, ao alterar o valor da variável global dentro da função, ela perde seu valor original e passa a ter o novo valor atribuído a ela. No exemplo acima, **a** deixa de valer 5 e passa a valer 8.

# Retornando Valores

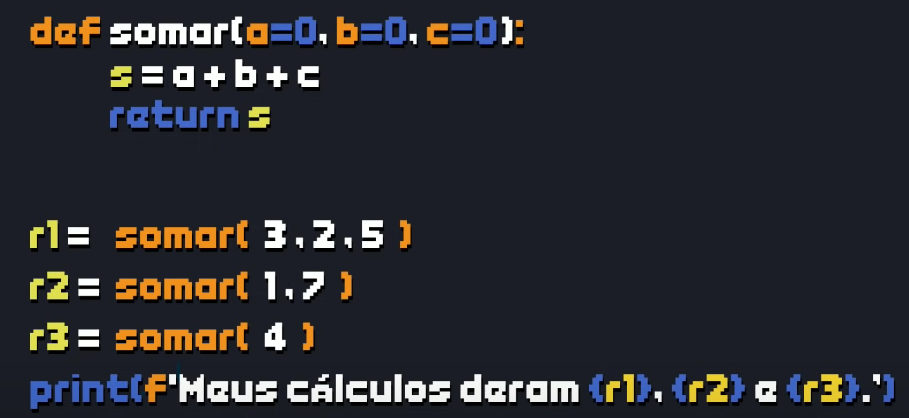
1. É possível retornar um valor para recuperá-lo e usar novamente ao invés de simplesmente imprimi-lo na tela. Para tanto, é preciso usar o comando **return** e colocar a chamada da **função** dentro de uma **variável** ou dentro de um **print** externo:



Ou

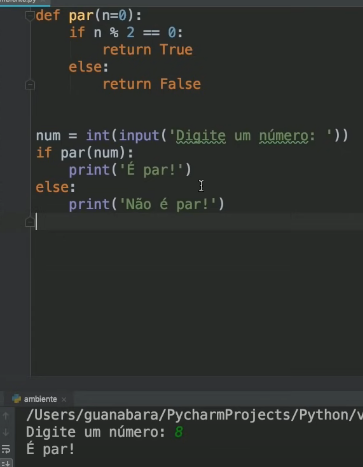


1. Com isso, é possível ter variáveis com resultados de somas diferentes:



* 1. Dessa forma — quando **a**, **b**, **c** forem **3**, **2**, **5** —, **r1** será o resultado dessa soma. E o mesmo irá acontecer com as novas variáveis e seus respectivos valores.

1. O retorno não é só para números. Pode retornar um booleano, texto, listas, tuplas etc. Exemplo de booleano:



# Modularização

1. Construção de módulos
2. Útil para:
   1. Dividir um programa grande em mais de um arquivo.
   2. Aumentar a legibilidade.
   3. Facilitar a manutenção.
   4. Organizar o código (dividir os problemas maiores em problemas menores).
   5. Ocultar o código detalhado.
   6. Reutilizar os módulos em outros projetos.

# Pacotes

1. São vários módulos separados em pastas.
2. A organização fica mais ou menos assim:
   1. uteis(pasta)
      1. Cores
      2. Datas
      3. Numeros
      4. Strings
   2. É importante que dentro de cada pasta exista um arquivo chamado “\_\_init\_\_.py”, como na imagem abaixo:
      1. 

# LISTAS

1. Um bom atalho:
   1. Idades\_proximo\_ano = [(idade + 1) for idade in idades]
      1. Vai criar uma lista somando 1 a cada idade da lista idades previamente criada.
      2. É um exemplo para se atualizar as idades dos usuários no ano seguinte.
2. Outro com exemplo:
   1. Print(idade for idade in idades if idade >21)
      1. Vai imprimir as idades que forem maiores que 21.
3. Quando fazemos uma tupla com uma lista dentro, podemos alterar os valores da lista. O que não podemos fazer é adicionar ou remover listas da tupla.

# ARRAY

1. As arrays do python são muito específicas, e servem, principalmente, para lidar com números.
   1. Em outros casos, o melhor é evitar usá-las.
      1. Em outras ocasiões, se quisermos trabalhar com números, o costume é usar a biblioteca NumPy.
2. As arrays do python devem ser importadas para o uso
3. Ex de uso de array:
   1. import array as arr  
      arr.array(‘d’, [1, 3.5])
      1. ‘d’ é um tipo específico para números float, conforme documentação das arrays.