

PYTHON

# INTRODUÇÃO (Aula 1 a aula 5)

**Python** é uma linguagem de programação de alto nível, interpretada, de script, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte. Foi lançada por Guido van Rossum em 1991. Atualmente possui um modelo de desenvolvimento comunitário, aberto e gerenciado pela organização sem fins lucrativos Python Software Foundation. Apesar de várias partes da linguagem possuírem padrões e especificações formais, a linguagem como um todo não é formalmente especificada.

## INSTALAÇÃO

O programa já vem nativo nas maiorias dos sistemas operacionais, com exceção do windows. Para a instalação, acesse o site oficial python: <http://python.org/> vá até a aba downloads, escolha a opção do seu sistema operacional e baixe a versão mais atual do python.

Para a instalação acione todas as permissões e adições PATH e siga a instalação normalmente. Desabilite as limitações do PATH do python no final da instalação.

Acesse o python pelo CMD apenas digitando python.

Acessando a pasta python, você também terá acesso ao IDLE.

## PyCharm

Para a instalação do PyCharm, você deve acessar o site oficial <https://jetbrains.com/pycharm/> clique no botão download, logo em seguida, escolha o seu sistema operacional e escolha a versão community para o uso livre.

Para a instalação, escolha a versão do seu sistema operacional e marque a opção para criar associação em .py depois é só clicar em avançar até concluir a instalação.

Para programar em um smartphone, baixe o app Qpython3.

## COMANDOS BÁSICOS

Comando para impressão: `print('o que você quer imprimir aqui')`

Comando para inserção de informação: `input('O que você quer inserir aqui')`

Para atrelar informações a alguma variável apenas coloque = depois da mesma. **Por exemplo:**  
`nome = input('Insira seu nome aqui: ')` ou `idade = input('insira sua idade aqui:')`

Você pode imprimir as informações das variáveis apenas usando o comando print e colocando o nome da variável, por exemplo: `print(nome)` ou `print(nome, idade)`.

## Referência (importante)

Para acessar as referências e obter dicas e tutorias de uso de vários comandos, módulos e etc, acesse o menu DOC no site oficial python.org e em seguida vá até a opção Library Reference, lá você encontra tudo o que você precisa.

**Mais de 680 recursos grátis para programador: <https://free-for.dev/>**

# AULA 6 e 7

## Tipos primitivos

Tipo Primitivo são os tipos de dados mais simples, isto é, a informação em sua forma mais primitiva. Bons exemplos de valores primitivos são os caracteres, os números, o valor **True** e o **False** e etc.

Os quatro tipos primitivos mais básicos que existem são:

**Int** = números inteiros (ex: 7, - 4, 0, 9875 etc)

**Float** = números reais ou números de ponto flutuante (ex: 4.5 , 0.076 , - 15.223 , 7.0 etc)

**Bool** = valores lógicos ou booleanos (ex: True ou False) sempre use maiusculo na primeira letra.

**Str** = valores caracteres ou strings (ex: 'Olá' , '7.5') todas as palavras tem que estar entre aspas que podem ser simples ou duplas.

Se o tipo primitivo não for especificado, ele automaticamente vira uma STR (string)

## Metódo

Para inserir informações dentro de um print, você pode usar os comandos:

Inserir = exemplo

```
Print('exemplo {}'.format(inserir))
```

```
Print(f'exemplo{inserir}')
```

Os dois comandos fazem a mesma função, porém o segundo é o mais simplificado e de fácil uso.

## Operadores aritméticos

**+** = adição (ex: 5 + 2 == 7)

**-** = subtração (ex: 5 - 2 == 3)

**\*** = multiplicação (ex: 5 \* 2 == 10)

**/** = divisão (ex: 5 / 2 == 2.5)

**\*\*** = potência (ex: 5 \*\* 2 == 25 | a potência equivale a 5<sup>2</sup>)

**//** = divisão inteira (ex: 5 // 2 == 2)

**%** = resto da divisão (ex: 5 % 2 == 1)

**==** = igual

**=** = recebe

## Ordem de precedência dos operadores

1 = ()                      3 = \* , / , // , %

2 = \*\*                     4 = + , -

## AULA 8

### Bibliotecas ou módulos

É usado para adicionar funcionalidades ao python, dentro do python você precisa utilizar o comando **import** para incluir essas funcionalidades

Exemplo: **math**, essa biblioteca adiciona vários comandos por exemplo:

**ceil** (faz um arredondamento numérico para cima)

**floor** (faz um arredondamento numérico para baixo)

**trunc** (elimina os números da virgula para frente sem fazer arredondamento).

**pow** (potência, funciona de forma semelhante ao **\*\***)

**sqrt** (calcula a raiz quadrada)

**factorial** (calcula o fatorial)

A partir do momento que eu utilizar o comando **import math** irá importar todas essas funcionalidades e muitas outras. Se você quiser utilizar apenas um desses módulos no seu código você pode utilizar o comando **from math import floor** por exemplo.

# AULA 9

## Manipulando textos

Dentro do python você pode fazer operações com strings. Exemplo:

```
frase = 'curso em vídeo'
```

Para usar o **fatiamiento**, apenas use o comando `frase[8]` por exemplo e você estará selecionando o caractere 'm', **lembrando** que os espaços também contam como caractere. Você também pode utilizar o comando `frase[:5]` para selecionar do caractere 5 até o início, ficando só a palavra 'curso', **lembrando** que fazendo este tipo de seleção o programa ignora o caractere digitado. Outro exemplo: `frase[15:]` começa do caractere 15 e vai até o final, como ignora o caractere digitado fica apenas 'ython'. Outro exemplo é o comando `frase[9::3]` que faz começar do caractere 9 e ir pulando de 3 em 3 caractere ignorando todos os outros.

Outras funcionalidades:

**len(frase)** = len vem de length que significa comprimento, esse comando mede em número a quantidade de caracteres. Ex: o len de frase é 21

**frase.count('o')** = Pede para contar quantas vezes o 'o', por exemplo, aparece e nesse exemplo ele aparece 3 vezes. Você pode alterar esse comando para **frase.count('o', 0, 13)** que significa que ele irá contar quantas vezes o 'o' aparece entre o caractere 0 e 13, lembrando que ele irá ignorar o caractere 13.

**frase.find('deo')** = Irá dizer onde encontrou o 'deo' dentro da frase. Se você colocar algum valor que não existe o comando irá te retornar o valor -1.

**'curso' in frase** = Irá dizer que existe a palavra 'curso' dentro de frase, indicando True ou False.

**frase.replace('python', 'android')** = Troca uma palavra pela outra escolhida, nesse exemplo, trocava a palavra python por android dentro da frase.

**frase.upper()** = Coloca todas as letras em maiúsculas dentro da frase nesse exemplo.

**frase.lower()** = Coloca todas as letras em minúsculas dentro da frase nesse exemplo.

**frase.capitalize()** = Coloca apenas a primeira letra em maiúsculo.

**frase.title()** = Coloca as letras do início de cada palavra em maiúscula.

**frase.strip()** = Remove todos os espaços inúteis dentro da string.

**frase.rstrip()** = Remove todos os espaços à direita dentro da string.

**frase.lstrip()** = Remove todos os espaços à esquerda dentro da string.

**frase.split()** = Cria uma divisão dentro da string considerando os espaços (exemplo: ['curso', 'em', 'video', 'python']).

**'-'.join(frase)** = Irá juntar todos os elementos da string e irá utilizar o '-' como separador (exemplo: 'curso-em-video-python')

## AULA 10 e 11

### Condições

Em uma condição, um ou outro comando será executado, nunca os dois ao mesmo tempo.

Por exemplo:

```
tempo = int(input('Quantos anos tem seu carro?'))
```

```
if tempo <= 3:
```

```
    print('carro novo')
```

```
else:
```

```
    print('carro velho')
```

```
print('—FIM—')
```

Ou seja, se a resposta dentro da variável for igual ou menor que 3 irá aparecer carro novo, senão, aparecerá carro velho.

Lembrando que todo comando que estiver colado no lado esquerdo será executado sempre e todo comando que estiver com a indentação para dentro (alinhado ao comando) ele pode ser executado ou não, vai depender da situação, no exemplo acima, ou ele vai escrever carro novo ou carro velho.

Nós também podemos fazer uma **CONDIÇÃO SIMPLIFICADA** utilizando o exemplo acima, nós poderíamos escrever o comando da seguinte forma:

```
tempo = int(input('Quantos anos tem seu carro?'))
```

```
print('carronovo' if tempo <= 3 else 'carro velho')
```

```
print('-fim-')
```

Esta forma mais simples deixa o programa mais simples, porém, cuidado pois pode deixar os códigos difíceis de entender para o programador. (Utilize a primeira fórmula para ter um programa mais limpo e claro).

### Cores no terminal

Para formatar as cores, estilo do texto é utilizado o comando \033[m, dentro desse comando você escolha por números as cores, estilo e cor do background. Por exemplo:

\033[0:31:44m , o número 0 representa o estilo da letra, o número 31 representa a cor e o número 44 representa a cor de fundo. Para retirar as cores e estilo basta colocar o \033[m

**Estilos:** 0 = nenhum , 1 = negrito , 4 = sublinhado , 7 = inverter as configurações

**Cores:** 30 = branco , 31 = vermelho , 32 = verde , 33 = amarelo , 34 = azul , 35 = roxo , 36 = ciano , 37 = cinza.

**Cores de fundo:** 40 = branco , 41 = vermelho , 42 = verde , 43 = amarelo , 44 = azul , 45 = roxo , 46 = ciano , 47 = cinza.

## AULA 12

### Condições aninhadas

Agora podemos criar várias condições e possibilidades. Nessa estrutura o comando else é opcional.

Por exemplo:

```
nome = str(input('Qual é seu nome?'))
```

```
if nome == 'Gustavo':
```

```
    print('Que nome bonito!')
```

```
elif nome == 'Pedro' or nome == 'Maria' or nome == 'Paulo':
```

```
    print('Seu nome é bem popular no Brasil.')
```

```
elif nome in 'Ana Claudia Jéssica Juliana':
```

```
    print('Belo nome feminino')
```

```
else:
```

```
    print('Seu nome é bem normal.')
```

```
print(f'Tenha um bom dia, {nome}!')
```

Como podemos ver no exemplo, agora podemos criar condições infinitas com o comando elif

# AULA 13

## Estrutura de repetição FOR

Com essa estrutura podemos criar repetições controladas com inicio e fim.

Por exemplo:

```
for c in range(1, 10):
```

```
    print('oi')
```

```
print('fim')
```

Nesse comando a palavra oi irá se repetir 10 vezes. (cuidado com as endentações)

A letra **c** na fórmula acima é utilizada da palavra contador, porém ela **pode ser substituída** por qualquer letra ou palavra da escolha do programador.

Se no exemplo acima for colocado o comando **print(c)** irá mostrar o número de 1 até 9, lembrando que o ultimo número é ignorado.

Para fazer uma **contagem regressiva** a fórmula fica assim: **for c in range (10, 0, -1)**, o -1 no final significa que vai começar no número 10 e vai tirar 1 até chegar no 0.

Se o comando for **for c in range (0, 7, 2)**, significa que irá começar do 0 e irá pulando 2 números até o número 7.

Podemos utilizar esse comando com uma **variável** também. Por exemplo:

```
i = int(input('Inicio: '))
```

```
f = int(input('Fim: '))
```

```
p = int(input('Passo: '))
```

```
for c in range (i, f + 1, p):
```

```
    print(c)
```

```
print('fim')
```

Lembrando que o + 1 utilizado na formula serve apenas para adicionar um número a mais porque que o python SEMPRE começa no número ZERO em TUDO.

Também é possível colocar um input dentro da formula for. Por exemplo:

```
for c in range(0, 10):
```

```
    n = int(input('Digite um valor: '))
```

Nesse comando, será pedido para adicionar um valor 10 vezes.

Também podemos colocar a variável **valor** dentro da estrutura for. Por exemplo:



# AULA 14

## Estrutura de repetição while

Com essa estrutura podemos criar repetições INFINITAS. A estrutura while serve tanto pra quando você sabe o limite ou não sabe o limite. Por exemplo:

```
c = 1
```

```
while c < 10:
```

```
    print(c)
```

```
    c = c + 1 OU c += 1
```

Com essa formula vamos obter o mesmo resultado do primeiro exemplo da repetição for, porém utilizamos mais linhas, como podemos observar o c vale 1 e enquanto a estrutura estiver repetindo o c vai receber + 1 até chegar ao 10 e parar a estrutura, como foi condicionado no while, enquanto o c for menor que 10 a estrutura repete.

Se você não souber a quantidade de vezes que o valor precisará ser inserido, não poderá utilizar a estrutura for, pois ela é limitada a um valor posto no programa. Por exemplo:

```
while n != 0:
```

```
    n = int(input('Digite um valor: '))
```

```
print('o programa terminou')
```

Como podemos observar nessa fórmula, enquanto o usuário não digitar o número ZERO ou seja, enquanto o número digitado for diferente de ZERO, a estrutura se repete INFINITAMENTE.

Também podemos colocar uma condição infinita da seguinte maneira:

```
r = 'S'
```

```
while r == 'S':
```

```
    n = int(input('Digite um valor: '))
```

```
    r = str(input('Quer continuar? [S/N] '))
```

```
print('acabou')
```

Agora enquanto o usuário não digitar a letra 'N' o programa irá continuar infinitamente.

# AULA 15

## Interrompendo repetições while

Como podemos ver, as repetições while são infinitas e para interrompe-las nós podemos utilizar o comando **break** e para isso também iremos utilizar a **estrutura de condições**. Por exemplo:

```
s = 0
```

```
while True:
```

```
    n = int(input('Digite um número: '))
```

```
    if n == 999:
```

```
        break
```

```
    s += n
```

```
print(f'A soma vale {s}')
```

No exemplo acima, enquanto o número 999 não for digitado o programa irá pedir um número infinitamente, o s colocado foi para demonstrar a soma de n no final do programa.

Quando usamos o comando break, ele sai do loop e não adiciona o número 999, já se fizermos com a fórmula while != 999 no final a soma iria acrescentar o número 999.

Nós também podemos utilizar as outras estruturas de repetições, tanto a simples só com o **if**, a composta com o **if e else** e as condições aninhadas com **if e elif**.

# AULA 16

## Tuplas

Tuplas são estruturas de variáveis compostas onde múltiplas informações são guardadas em grupo na memória.

Tuplas possuem as seguintes características: São definidas por **( )** , são **imutáveis** enquanto o programa estiver em execução.

Nas versões mais recentes do Python, não é necessário usar **()**

### Exemplos:

```
lanche = ("Hamburger", "Batata", "Cenoura", "Banana")
```

```
print(lanche) # mostra a tupla inteira
```

```
print(lanche[1]) # mostra o elemento 1 da tupla, lembrando que o python sempre começa em 0
```

```
print(lanche[-2]) # mostra o penultimo elemento
```

```
print(lanche[1:3]) # mostra o intervalo fatiado [1:3], nota-se que o elemento 0 fica de fora e o elemento 3 também fica de fora também, pois o intervalo vai de 1 a 3, sem incluir o 3.
```

```
print(lanche[-3:]) # mostra o antepenultimo até o final
```

Lembre-se que tuplas são imutáveis enquanto o programa estiver em execução. Remova o comentário da linha seguinte para verificar o erro. As **#** foram colocadas em forma de comentário para melhor entendimento do conteúdo.

### Loop for para tuplas

Utilizando o mesmo exemplo acima, nós também podemos fazer loop com tuplas utilizando a repetição for. Por exemplo:

```
for comida in lanche:
```

```
    print(f'Eu vou comer {comida}')
```

Este comando irá mostrar todas as palavras dentro da tupla lanche.

Para mostrar a **posição dos elementos da tupla** podemos fazer da seguinte forma:

```
for contador in range(0, len(lanche)):
```

```
    print(f'Eu vou comer {lanche[contador]} na posição {contador}')
```

Ou nós também podemos utilizar o seguinte comando:

**for posição, comida in enumerate(lanche):**

```
    print(f'Eu vou comer {comida} na posição {posição}')
```

Nós podemos utilizar o comando **sorted(lanche)** para organizar a tupla em ordem alfabética

O comando **lanche.index()** podemos verificar qual posição o valor colocando dentro de **()** está.

Você pode apagar uma tupla inteira com o comando **del(lanche)**.

# AULA 17

## Listas

Listas, assim como as tuplas, são estruturas de variáveis compostas onde múltiplas informações são guardadas em grupo na memória.

Listas possuem as seguintes características: São definidas por `[ ]` ou `list()` , são **mutáveis** enquanto o programa estiver em execução.

### Comandos e exemplos de listas:

#### Adição de valores:

`lanche.append('valor')` **#inclui valor na última posição**

`lanche.insert(0,'valor')` **#substitui valor na posição '0'**

#### Remoção de valores:

`del lanche[3]` **#elimina o valor na posição '3'**

`lanche.pop()` **#elimina o último e reposiciona os valores na lista**

`lanche.remove('valor')` **#elimina o valor indicado**

`if 'valor' in lanche:` **#evita mensagem de erro no comando remove**

`lanche.remove('valor')` **#evita msg de erro**

#### Manipulação da lista:

`lista2 = list(range(4,11))` **# Cria uma estrutura organizada, resposta: lista2 = [4,5,6,7,8,9,10]**

`lista3 = [8,5,4,3,0]`

`lista3.sort()` **#ordena os valores, resposta: [0,3,4,5,8]**

`lista3.sort(reverse=True)` **# Ordena os valores de forma inversa, resposta: [8,5,4,3,0]**

`len(lista3)` **# Conta quantos elementos tem dentro da lista, resposta: 5**

Resumão Comentado

`num = [2,5,9,1,7]`

`num1 = num` **# Faz uma conexão entre lista num e num1**

`num1 = num[:]` **# Gera uma cópia de num**

`#num.append(8)` **#adiciona '8'**

`#num[0] = 3` **#substitui a posição 0 pelo valor '3'**

`#num.sort(reverse=True)` **#ordena de forma inversa**

`#num.insert(2,4)` **#adiciona valor '4' após a posição 2 reordenando a lista**

`#num.pop()` **#exclui o último valor '8'**

`#num.pop(4)` **#exclui o valor na posição 2 por '1'**

# AULA 18

## Listas parte 2

Nas listas podemos inserir **varias listas** dentro de uma única lista. Por exemplo:

```
dados = list()
```

```
personas = list()
```

```
dados.append('Pedro')
```

```
dados.append(25)
```

Dentro da lista de dados temos duas informações 'Pedro' e 25, e podemos colocar ela dentro da lista personas como uma única informação utilizando o comando [:] para fazer uma cópia. Exemplo:

```
personas.append(dados[:])
```

Agora dentro da lista persona temos a lista dados na posição zero e podemos inserir quantas listas quisermos nas posições seguintes normalmente.

Nós podemos criar várias estruturas dentro de uma lista abrindo mais um [ ], como por exemplo:

```
personas = [['Pedro', 25], ['Maria', 19], ['João', 32]]
```

Ou seja, agora dentro da lista personas nós temos mais 3 listas inseridas.

**Podemos extrair informações das listas dentro da lista** utilizando o comando, por exemplo, **print(personas[0][0])** nesse comando irá mostrar o nome Pedro que é a primeira informação da primeira lista.

**print(personas[1][1])** irá mostrar 19 que é a segunda informação dentro da segunda lista

**print(personas[2][0])** irá mostrar João que é a primeira informação dentro da terceira lista.

**print(personas[1])** irá mostrar toda a informação dentro da segunda lista ['Maria', 19]

O comando **clear** limpa os valores dentro de uma lista, por exemplo, **dados.clear()**.

# AULA 19

## Dicionarios

Os dicionarios são estruturas de dados semelhantes as tuplas e as listas, só que dessa vez conseguimos ter indices literais, por exemplo, ao invés de usarmos numeros 0, 1, 2... Nós podemos nomear esses indices. Os dicionarios em python são identificados por `{ }`, ou então por `dados = dict()`.

Utilizando `dados =` como um exemplo.

Nós podemos utilizar o dicionario para nomear da seguinte maneira:

```
dados = {'nome': 'Pedro', 'idade': '25'}
```

Agora ao invés de colocarmos `print(dados[0])`, nós podemos utilizar o comando `print(dados['nome'])` e com esse comando irá mostrar a informação 'Pedro'.

Utilizando o mesmo exemplo, podemos utilizar o comando `print(dados['idade'])` e irá mostrar a informação '25'.

Se quisermos **adicionar** alguma informação dentro do dicionário, **não podemos utilizar o comando `append()`**. Para isso podemos utilizar o comando, `dados['sexo'] = 'M'` e com esse comando irá adicionar o indice sexo ao final do dicionario com a informação 'M'.

Para **remover** algum elemento dentro de um dicionario, podemos utilizar o comando `del dados['idade']` seguindo o exemplo acima, ele irá deletar o indice idade do dicionario.

É **importante** entender a diferença entre valores, chaves e itens, os **valores** são as informações dentro do dicionario, **chaves** são os indices dentro do dicionario e os **itens** são as informações valores e chaves juntas. Por exemplo:

```
print(dados.values()) # Esse comando irá mostrar todos os valores.
```

```
print(dados.keys()) # Esse comando irá mostrar todas as chaves.
```

```
print(dados.items()) # Esse commando irá mostrar todos os itens.
```

Nós podemos utilizar esse conceito de valores, chaves e itens, nos laços, como por exemplo, na estrutura for.

### Exemplo:

```
filme = {'titulo': 'Star Wars', 'ano': '1977', 'diretor': 'George Lucas'}
```

```
for k, v in dados.items():
```

```
    print(f'O {k} é {v}')
```

Nesse comando irá mostrar: **O titulo é Star Wars**. A letra **k** representa as chaves e a letra **v** representa os valores. A estrutura for é de repetição, ou seja, seguindo esse exemplo, ele irá mostrar logo em seguida: **O ano é 1977** feito isso, ele irá voltar mais uma vez o loop e irá mostrar em seguida: **O diretor é George Lucas**, chegando ao final, ele não tem mais nenhum elemento, então, ele termina o laço, termina a estrutura de repetição.

Nos dicionarios não podemos utilizar o enumerate como nas listas e tuplas.

Você pode juntar listas, tuplas e dicionarios. Por exemplo, você pode criar uma lista chamada locadora e utilizar o comando `locadora.append(filme)` sendo assim o dicionario irá ficar no indice

0 dentro da lista locadora, e você pode colocar quantos dicionários quiser dentro da lista. Com essa lógica então você pode utilizar o comando **print(locadora[0]['ano'])**, o ZERO é a referência externa, ou seja, o dicionário com todas as informações dentro, e o ano é a informação 1977 dentro da referência 0, logo, irá mostrar: 1977.

### Outro exemplo:

```
brasil = []

estado1 = {'uf': 'Rio de Janeiro', 'sigla': 'RJ'}
estado2 = {'uf': 'São Paulo', 'sigla': 'SP'}

brasil.append(estado1)
brasil.append(estado2)
```

Se você utilizar **print(estado1)** irá mostrar {'uf': 'Rio de Janeiro', 'sigla': 'RJ'} se utilizar o **print(estado2)** irá mostrar {'uf': 'São Paulo', 'sigla': 'SP'} e se utilizar o **print(brasil)** irá mostrar os dois elementos dentro da lista: [{'uf': 'Rio de Janeiro', 'sigla': 'RJ'}, {'uf': 'São Paulo', 'sigla': 'SP'}] Se você utilizar **print(brasil[0])** irá mostrar o estado que foi adicionado primeiro, ou seja, irá aparecer {'uf': 'Rio de Janeiro', 'sigla': 'RJ'} a mesma lógica se aplica utilizando o comando **print(brasil[1])**, agora, se você utilizar o comando **print(brasil[0]['uf'])** irá mostrar somente a informação Rio de Janeiro.

Você pode adicionar informações dentro de um dicionário e em seguida dentro de uma lista, da seguinte maneira:

```
estado = dict()

brasil = dict()

for c in range(0, 3):

    estado['uf'] = str(input('Unidade Federativa: '))
    estado['sigla'] = str(input('Sigla do Estado: '))

    brasil.append(estado.copy())

print(brasil)
```

Dentro dos dicionários, não pode haver fatiamento, por isso não é possível copiar utilizando o comando **[:]**, para copiar é necessário utilizar o comando **.copy()** como no exemplo acima.

Para deixar organizado a forma de exibir as informações, nós podemos utilizar uma **estrutura de repetição for**, dentro de outra estrutura de repetição for, para isso, basta retirar o comando no final da lista **print(brasil)** e seguir o exemplo abaixo:

for e in brasil:

```
    for k, v in e.items():
        print(f'O campo {k} tem valor {v}.')
```

Irá aparecer: O campo uf tem valor Rio de Janeiro, por exemplo.

**Também podemos fazer a seguinte forma:**

for e in brasil:

```
    for v in e.values():
        print(v, end=' ')
```

E irá aparecer: Rio de Janeiro RJ, por exemplo, e em seguida os outros estados digitados. Ou seja, você pode utilizar laços dentro de laços.

## AULA 20

### Funções

Funções são basicamente **rotinas** dentro de um programa, o python já vem com muitas funções dentro dele, por exemplo, **print()**, **len()**, **input()**, **int()**, **float()**, todos esses comandos são funções, mas essas funções existentes nem sempre são as que nós satisfazem e precisamos criar novas funções dentro do programa.

Outro exemplo é se nós repetimos sempre alguma linha de código ou comando dentro do programa, ou seja, **uma rotina**, e o python não tem uma função específica para facilitar, então, para não precisar digitar a linha inteira toda vez, nós podemos criar uma nova função.

Se em um programa, você sempre mostra traços para fazer uma divisão de linha a cada menu, por exemplo, você torna repetitivo e cansativo a digitação do comando, pra isso nós podemos criar a função a seguir:

```
def mostrarLinha():
```

```
    print('-----')
```

Agora a cada vez que precisar utilizar essa linha, ao invés de digitar o código, podemos utilizar apenas o comando **mostrarLinha()**

### Funções com parametros

Podemos **adaptar** ainda mais as funções a nossa necessidade usando **parametros**. Se quisermos inserir uma mensagem diferente entre duas linhas cada vez que usarmos uma função, por exemplo, nós podemos utilizar o seguinte comando:

```
def mensagem(msg):
```

```
    print('-----')
```

```
    print(msg)
```

```
    print('-----')
```

A partir de agora nós podemos criar uma chamada assim: **mensagem('SISTEMA DE ALUNOS')** a mensagem SISTEMA DE ALUNOS será copiada direto para o parametro **msg** e será exibida entre as duas linhas assim que o programa for executado, agora nós podemos criar várias chamadas com diferentes mensagens de uma forma muito mais simples.

Podemos definir calculos utilizando este principio, ao invés de utilizar o comando:

```
a = 1
```

```
b = 2
```

```
s = a + b
```

```
print(s)
```

Nós simplesmente poderíamos criar uma função.

```
def soma (a, b):
```

```
    s = a + b
```

```
    print(s)
```

```
soma(1, 2)
```

```
soma(2, 3) #etc...
```

Ao invés de repetirmos a primeira fórmula várias vezes trocando apenas os números, nós podemos simplesmente fazer a chamada soma(1, 2) quantas vezes quisermos com vários números e somas diferentes.



## Desempacotamento

Com essa funcionalidade, nós podemos utilizar vários **parametros**, diferente do exemplo anterior que só utilizamos 2, o parametro a e b. Para isso basta utilizar o comando **\*** na frente do parametro, por exemplo:

```
def contador(*núm):
```

```
    print(num)
```

```
contador(2, 1, 7)
```

```
contador(8, 0)
```

```
contador(4, 4, 7, 6, 2)
```

Com esse comando, nós acabamos de criar uma **tupla** e nós podemos trabalhar com as mesmas funcionalidades de uma **tupla**, podemos utilizar comandos de for para tuplas dentro da funcionalidade também, por exemplo:

```
def contador(*num):
```

```
    for valor in num:
```

```
        print(valor, end='')
```

```
    print('FIM!')
```

```
contador(2, 1, 7)
```

```
contador(8, 0)
```

```
contador(4, 4, 7, 6, 2)
```

Com essa funcionalidade irá mostrar os valores que estão dentro da tupla, formatado com a **estrutura de repetição for** e no final de cada tupla irá aparecer a palavra **FIM!**

Podemos utilizar o comando **len()** também para medir quantos números tem dentro das tuplas, embaixo da estrutura def poderíamos acrescentar **tam = len(num)** e em seguida **print(f'Recebi os valores {num} e são ao todo {tam} números')** e com isso iria mostrar as tuplas e em seguida quantos números existem dentro de cada tupla.

## Listas

As tuplas são limitadas por serem imutaveis, porém isso não é um problema, com esse metodo também podemos trabalhar com as **listas**. Por exemplo:

```
def dobra(lst):
```

```
    pos = 0
```

```
    while pos < len(lst):
```

```
        lst[pos] *= 2
```

```
        pos += 1
```

```
valores = [7, 2, 5, 0, 4]
```

```
dobra(valores)
```

```
print(valores)
```

Tudo que for feito com o parametro **lst** irá afetar a lista, na formula acima podemos ver que enquanto a **pos**(posição) for menor do que a lista (ZERO) ele irá dobrar os valores de cada posição dentro da lista.

# AULA 21

## Funções (Parte 2)

**Interactive Help** – Serve para obter ajuda interativa no python vá até o **Python Console** e utilize a função interna **help()**. Lá dentro você tem um manual completo Python, para sair digite **quit**. Dentro do ambiente python também é possível utilizar o comando, porém terá que ser de uma forma mais direta, por exemplo: **help(print)** irá mostrar tudo sobre a funcionalidade **print**.

**Docstrings** – Serve para colocar um manual de uma função **criada**, para isso basta colocar aspas duplas três vezes logo depois da função criada, por exemplo:

```
def contador(i, f, p):  
    """  
    -> Faz uma contagem de mostra na tela.  
    :param i: início da contagem  
    :param f: fim da contagem  
    :param p: passo da contagem  
    :return: sem retorno  
    """  
    c = i  
    while c <= f:  
        print(f'{c}', end='..')  
        c += p  
    print('FIM!')
```

Agora para obter o manual colocado acima basta usar o comando **help(contador)**, e você irá obter o manual colocado logo abaixo da função.

**Parâmetros opcionais** – É quando você coloca uma opção dentro de uma variavel quando lhe não for atribuido um valor. Por exemplo:

```
def somar(a=0, b=0, c=0):  
    s = a + b + c  
    print(f'A soma vale{s}')  
somar(3, 2, 5)  
somar(8, 4)  
somar()
```

Como podemos observar, se não fosse atribuido um **parametro opcional** no exemplo acima, o comando não iria funcionar, pois na segunda opção não foi atribuido o valor **C**, e na terceira opção não foram atribuidos valores **A**, nem **B** e nem **C**. Como foi atribuido o parametro **0**, irá funcionar normalmente.

**Escopo de variáveis** – Toda variavel criada dentro de uma função é um **escopo local**, e só funciona dentro da função, já toda variavel criada fora de uma função é um **escopo global** e funciona em toda estrutura. No exemplo abaixo o **B** recebe o valor de **A** ditado no escopo global.



Para fazer a variável **A** dentro da função ser global, teríamos que digitar o comando **global a**, e ficaria assim:



Agora a variável **A** dentro da função passa a funcionar em toda a estrutura, e a variável **A** fora da função **perdeu** o valor **5** que tinha anteriormente.

**Retorno de valores** – As funções dentro do python, podem ou não retornar um valor e pra isso usamos o comando **return**. Com essa comando podemos mostrar os resultados dentro da função da maneira que quisermos, dando uma formatação desejada **ao invés de** mostrar repetidamente a mesma coisa mudando somente os valores. **Exemplo:**

```
def somar(a=0, b=0, c=0):  
    s = a + b + c  
    return s  
  
r1 = somar(3, 2, 5)  
r2 = somar(1, 7)  
r3 = somar(4)  
print(f'Meus cálculos deram {r1}, {r2} e {r3}.')
```

# AULA 22

## Modulos e Pacotes

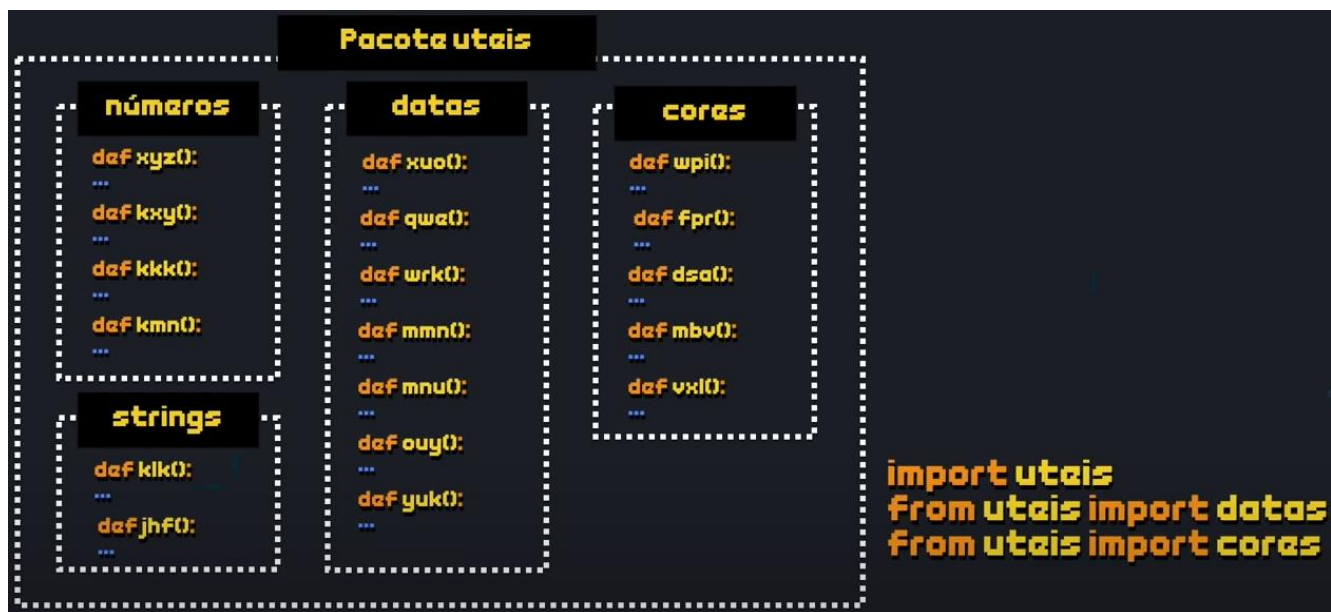
Modularização serve para dividir um programa grande em **modulos** aumentando a legibilidade e facilitando a sua manutenção.

Para **criar modulos** basta criar um novo arquivo **.py** com as funcionalidades que você deseja separar do programa em seguida volte ao programa e utilize o comando **import**. Por exemplo, você pode criar um arquivo chamado **uteis.py** e para importar ele no seu programa utilize o comando **import uteis** e para utilizar as funções dentro desse modulo utilize o comando **uteis.função()** no lugar da função, você digita a **função** que deseja.

*Obs.: Não esqueça de usar o **return** dentro da função quando for utilizar os módulos*

**Vantagens de usar modulos:** Organização do código, ocultação de código detalhado, facilidade na manutenção e a reutilização em outros projetos.

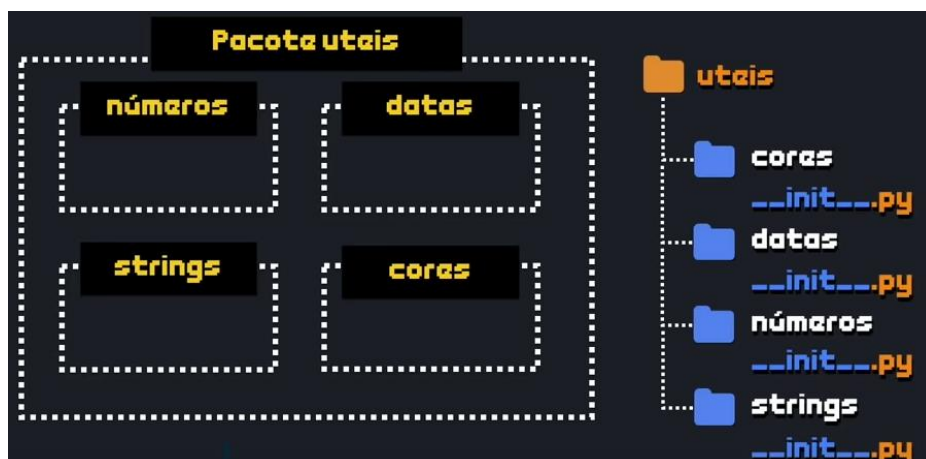
Os **pacotes** servem para quando os modulos ficam grandes demais, organizando esses modulos em vários assuntos dentro de um pacote. Por exemplo:



No exemplo acima podemos ver que dentro de **pacote uteis**, temos vários módulos organizados por nomes, e para utilizá-los basta utilizar o comando **import**, como no exemplo a direita da imagem.

Para **criar um pacote**, basta criar uma pasta dentro do projeto (Python Package).

Dentro de um pacote você pode criar vários pacotes e os modulos deveram ser nomeados como **\_\_init\_\_.py** dentro de cada pacote, lembrando que utilizar pacotes é recomendado apenas se for um projeto **GIGANTESCO**, caso contrário, apenas utilizar os módulos já resolve. **Exemplo:**



## AULA 23

### Tratamento de erros e exceções


O Python permite tratar erros e criar respostas a essas exceções, dentro do Python existe infinidades de exceções, por exemplo:

A vertical list of Python exception names in orange text on a black background. The exceptions listed are: NameError, ValueError, ZeroDivisionError, TypeError, IndexError, KeyError, EOFError, KeyboardInterrupt, OSError, MemoryError, ConnectionError, and RuntimeError.

**NameError**  
**ValueError**  
**ZeroDivisionError**  
**TypeError**  
**IndexError**  
**KeyError**  
**EOFError**  
**KeyboardInterrupt**  
**OSError**  
**MemoryError**  
**ConnectionError**  
**RuntimeError**

Essas são somente algumas exceções, e dentro do programa Python é representado por **Exception**, exceções não são erros sintáticos, ou seja, você digitou o comando certo, porém acontece algum erro, por exemplo, falta de variável, divisão por zero, etc.

Para contornar isso podemos utilizar o comando **try**, **except** e **else**, como por exemplo:

A flowchart illustrating the try-except-else logic. It starts with 'try:' in orange, followed by a blue box containing 'operação'. Then 'except:' in orange, followed by a blue box containing 'falhou'. Finally 'else:' in orange, followed by a blue box containing 'deu certo'.

**try:**  
**operação**  
**except:**  
**falhou**  
**else:**  
**deu certo**

Dentro do **try** colocamos a operação e se der algum erro, nós podemos executar algo dentro do **except** e se der certo nós podemos executar o **else**. Com isso a mensagem de erro não acontece.

Exemplo em código prático:

```
try:
    a = int(input('Numerador: '))
    b = int(input('Denominador: '))
    r = a / b
except:
    print('Infelizmente tivemos um problema!')
else:
    print(f'O resultado é {r}')
```



Nós também podemos utilizar o comando **finally**, esse comando serve para ser executado independente se o código der **certo ou der falha**. Por exemplo:

```
try:
    a = int(input('Numerador: '))
    b = int(input('Denominador: '))
    r = a / b
except:
    print('Infelizmente tivemos um problema!')
else:
    print(f'O resultado é {r}')
finally:
    print('Volte sempre! Muito obrigado!')
```

No comando acima a frase Volte sempre! Muito obrigado, será executada independente se o código der certo ou errado.



Todo **try** pode ter mais de um **except**, cada um deles com tipos diferentes de exceção e cada um deles terá seu próprio bloco com sua própria mensagem, com seu próprio tratamento.

Exemplo em código prático:

```
try:
    a = int(input('Numerador: '))
    b = int(input('Denominador: '))
    r = a / b
except (ValueError, TypeError):
    print('Tivemos um problema com os tipos de dados que você digitou.')
except ZeroDivisionError:
    print('Não é possível dividir um número por zero!')
except KeyboardInterrupt:
    print('O usuário preferiu não informar os dados!')
else:
    print(f'O resultado é {r}')
finally:
    print('Volte sempre! Muito obrigado!')
```

Você também pode verificar o tipo de erro utilizando o seguinte comando:

```
except Exception as erro:
    print('O erro encontrado foi {erro.__cause__}')
```