Seção 3: Python Intermediário

▼ Funções

- Definido pelo comando *def* e o nome da função (sempre apresentando os parênteses após o nome);
- Funções podem receber parâmetros, que são variáveis que serão usadas dentro do código da função; Por padrão, se a função está sendo criada para receber algum parâmetro, se ele não for passado no código, um erro irá acontecer e parar o programa. Para contornar isso é possível usar valores padrões nos parâmetros, ou seja, caso não seja passado o parâmetro, a variável da função irá receber outro valor pré escrito; Funções podem receber mais de um parâmetro, sendo eles separados por vírgula;
- Caso o argumento receba um valor padrão, argumentos à frente dele também precisarão de valores padrão.
- Os parâmetros possuem uma ordem bem definida, então na chamada da função eles irão obedecer essa ordem. Caso necessário, é possível informar os parâmetros fora de ordem, mas pra isso é necessário informar o nome da variável e o dado que ela está recebendo;
- Se um argumento for chamado pelo nome da variável, se outro argumento for passado logo à frente, esse também precisará ser chamado pelo nome da variável.
- Funções podem retornar valores através do comando *return*. Para que não haja nenhum erro, no código uma variável deve receber a função, assim a variável vai ter o valor retornado pela função; Se uma variável está recebendo uma função que não possui o return (ou possui mas não está retornando nada, ou seja, somente o comando *return* está escrito), o código da função vai ser executado normalmente e a variável vai receber um valor *None* (tipo de dado que significa “sem valor”); Sempre que no código um *return* for encontrado, tudo que estiver abaixo dele não será executado;

```
# Exercício 1
# Criar uma função saudação que exibe uma saudação com parâmetros saudação e nome.
#
# def saudacao(ola='Olá', nome='desconhecido'):
#     print(f'{ola}, {nome}')
#
#
# saudacao('Bom dia', 'Andre')
# saudacao()
# saudacao('Boa tarde', 'Robson')
#

# ------------------------------------------------------------------------------------#
# Exercício 2
# Crie uma função que recebe 3 números com parâmetros e exiba a soma entre eles.

# def soma(n1=0, n2=0, n3=0):
#     print(f'A soma entre {n1}, {n2} e {n3} é igual à: {n1 + n2 + n3}')
#
#
# soma(2, 5, 6)
# soma(1, 1, 1)
# soma()
# soma(5, 6, 7)

# ------------------------------------------------------------------------------------#
# Exercício 3
# Crie uma função que receba 2 números. O primeiro é um valor e o segundo um percentual.
# Retorne o valor do primeiro número somado do aumento do percentual do mesmo.

# def aumento(valor, percentual):
#     return valor + (valor * (percentual / 100))
#
#
# print(f'O valor de 200 com aumento de 40% é igual à: {aumento(200, 40)}')

# ------------------------------------------------------------------------------------#
# Exercício 4
# Fizz Buzz - Se o parâmetro da função fo divisível por 3, retorne fizz. Se o parâmetro da função for
# divisível por 5, retorne buzz. Se o parâmetro da função for divisível por 5 e por 3, retorne FizzBuzz.
# Caso contrário, retorne o número enviado.

def fizzbuzz(num):
```

```
    if num % 5 == 0 and num % 3 == 0:
        return 'FizzBuzz'
    if num % 3 == 0:
        return 'fizz'
    if num % 5 == 0:
        return 'buzz'
    return num


print(fizzbuzz(15))
```

- Quando não se sabe quantos parâmetros serão passados para uma função, é possível fazer um empacotamento de argumentos. Isso é feito nomeando a variável do argumento normalmente mas adicionando um asterisco (*) na frente do nome. Fazendo isso, uma tupla com os valores vai ser criada. Por convenção da comunidade python, quando isso acontecer a variável se chama “args”.
- Semelhantes a esse empacotamento de argumentos, existe o empacotamento de argumentos nomeados. Por convenção, essa variável recebe o nome de *kwargs*. Além disso, ela possui dois asteriscos na frente do nome. Assim, os argumentos que vão ser passados na função deverão ser nomeados e, agora invés de criar uma tupla com os valores, será criado um dicionário (nome e valor).
- Escopo de variáveis:
    - Variáveis criadas fora do escopo de alguma função, essa terá escopo global. Isso significa que ela terá o mesmo valor e poderá ser usada em qualquer parte do programa. Se ela for alterada dentro de uma função, ela será alterada somente dentro dessa função (passa a ser uma variável de escopo local);
    - Caso seja necessário alterar por definitivo o valor de uma variável, é necessário usar o comando *global* e escrever o nome da variável logo em seguida. Porém isso não é considerado uma boa prática de programação;
    - Variáveis locais de uma variável não pode ser usado em outra função;

▼ Expressões Lambda (funções anônimas)

- Usa-se muito quando é necessário passar uma função para outra, pois se ela não precisar de muitos códigos isso poupa linhas, ou seja, não é preciso criar toda uma função somente para isso. Exemplo:

```
# Exemplo de uma função simples
def multi(x, y):
  return x * y

var = multi(9, 9)
print(var)

>>> 81


# Com expressões lambda
multi = lambda x, y: x * y
print(multi(9, 9))

>>> 81

# É possível ordenar listas com listas dentro também

lista = [['P1', 9], ['P2', 4], ['P3', 7], ['P4', 2]]
# Se somente tentar usar o método sort de lista, o resultado será exatamente igual a lista atual. Porém, se usar uma expressão lam

lista.sort(key=lambda, item: item[1])
print(lista)

>>> [['P4', 2], ['P2', 4], ['P3', 7], ['P1', 9]] #parâmetro reverse=True pode ser usado para ordenar de forma decrescente
```

▼ Tuplas

- Semelhante à listas, porém elas não podem ser alteradas, ou seja, nenhum dado pode sofrer alterações; Os itens só podem ser adicionados ou tirados na sua criação;
- É criada entre parênteses; Pode ser criada sem eles, somente separando por vírgula;
- Seus índices obedecem a mesma regra;
- O fatiamento também segue a mesma regra;

- Tuplas aceitam concatenação;
- Se uma tupla for multiplicada por um número inteiro, ela irá se repetir esse número de vezes. É o mesmo resultado de multiplicar uma string por algum número inteiro;
- Para converter uma tupla para uma lista basta usar o comando list( );

▼ Dicionários

- Semelhante à listas, apresenta índice e um valor. Porém, nos dicionários, o programador controla o índice (chave);
- É criado entre chaves;

```
# Exemplo de criação de um dicionário
dicionario = {'chave1': 'valor da chave'}

# Para criar novas chaves com valores, basta usar o identificador de índice e apontar a nova chave
dicionario['nova_chave'] = 'Valor da nova chave'
```

- Se for criado mais de uma chave com mesmo nome, a última é que será levado em conta para representar o valor da chave;
- Para atualizar o valor de uma chave, basta atribuir à chave um novo valor, isso faz com que seu valor mude;
- O comando del também é usado para excluir valores de um dicionário, mas é necessário indicar a chave;
- Iterar sobre um dicionário com um laço FOR, por padrão, faz com que somente o laço percorra sobre as chaves. Caso queira acessar os valores, é usado a função .values( ). Para acessar os dois, é usado a função .items( );
- Atribuir um dicionário à uma variável não faz com que a variável crie um novo dicionário idêntico, faz com que uma ligação entre eles seja feito. Ou seja, se alterar algum valor ou chave da variável, alterará no dicionário originar também; Caso seja necessário, para fazer isso é possível usar a função .copy( ), porém é uma cópia rasa, ou seja, eles ainda têm uma certa ligação;
- Concatenar dicionários é possível e é feito com a função .update( ), passando como parâmetro o dicionário que se deseja concatenar com o outro.

▼ Sets (conjuntos)

- Também é uma estrutura de dados composta, ou seja, que agrupa vários dados em um só local. Porém, a diferença com as outras estruturas é que os conjuntos só suportam elementos únicos, ou seja, não aceita elementos repetidos;
- Sua criação também é feita através de chaves { }, porém sem as keys que nomeiam os índices;
- Eles são iteráveis, porém não possuem índices, então não é possível acessar valores através de colchetes [ ];
- Alguns métodos e operadores que são utilizadas em conjuntos:
  - .add( ): adiciona um novo elemento ao conjunto;
  - .update( ): funciona semelhante a função add, porém se for passado como parâmetro uma string, essa string será iterada e cada elemento será colocado separadamente no conjunto (caso a string esteja dentro de uma lista ou outra estrutura composta, a string será adicionada por completo ao conjunto), o mesmo acontece com listas e outras estruturas compostas;
  - .discard( ): exclui um elemento do conjunto;
  - .clear( ): remove todos os elementos de um conjunto;
  - | (pipe): une conjuntos;
  - & (intersection): retorna a interseção entre conjuntos;
  - - (diferença): a ordem dos conjuntos faz diferença aqui. Esse operador só vai analisar o conjunto da esquerda, retornando somente o elemento que não está presente no conjunto da direita;
  - ^ (diferença simétrica): retorna somente os valores únicos dos conjuntos, exemplo:

```
s1 = {1, 2, 3, 4, 5, 6, 8}
s2 = {1, 2, 3, 4, 5, 6, 7}
s3 = s1 ^ s2
print(s3)

>>> {7, 8}
```

- Os conjuntos podem ser interessantes para remover elementos duplicados de uma lista, bastando fazer um cast da lista para um conjunto com a função set( ) (geralmente os elementos voltarão fora de ordem).

▼ List Comprehension em Python (compreensões da listas)

- Usado para escrever menos linhas de códigos e otimizar o programa (performance);

Exemplo:

```
lista = [1, 2, 3, 4, 5, 6, 7]
multi = [variavel * 2 for variavel in lista] # ou seja, a variavel está recebendo o valor de cada elemento da lista, multiplicado

print(multi)

>>> [2, 4, 6, 8, 10, 12, 14]
```

- é possível usar condicionais dentro das lists comprehensions;

Outro exemplo:

```
numeros = list(range(100))
teste = [v for v in numeros if v % 2 == 0 if v % 3 == 0]
print(teste) # só irá mostrar os valores divisíveis por 2 e 3 ao mesmo tempo, é como se tivesse um operador AND junto dos IF's

>>> [0, 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78, 84, 90, 96]

# caso o uso do ELSE seja necessário, as condicionais deve m vir antes do FOR (aí se quiser pode usar operadores como AND, OR, etc
```

▼ Dictionary Comprehensions em Python (compreensão de dicionários)

- A criação é semelhante à compreensão de listas, porém agora com a utilização de chaves { };
- Sempre será passado o par de chave e valor seguindo o exemplo: {x : y for x, y in lista}, ou seja, está sendo criado uma chave e um valor para cada chave e valor presente em uma lista (ou outra estrutura composta);
- Se for somente para formar um dicionário é possível usar a função *dict.* Porém, usando compreensão de dicionário, é possível usar algumas funcionalidades a mais, tipo usar alguma função de string (upper, lower, etc.);
- Se não for colocado os dois pontos (:) para indicar chave e valor, é criado um conjunto (set);

▼ Geradores, Iteradores e iteráveis em Python

- Iteráveis em python são objetos que contém o método **iter**, ou seja, que aceita por exemplo que um laço FOR seja executado sobre ele (percorrer cada item). Para conferir se um objeto é iterável, é possível usar a seguinte linha de código:

```
lista = [1, 2, 3, 4, 5, 6, 7, 'blabla']
print(hasattr(lista, '__iter__')) # Isso irá retornar False or True
```

O que acontece é que quando se itera sobre um objeto, de forma temporária o objeto se transforma em um iterador. Um iterador pode retornar somente um valor de cada vez e seu comportamento pode ser visto abaixo:

```
lista = [1, 2, 3, 4, 5, 6]
lista = iter(lista)
print(next(lista))
>>> 1
print(next(lista))
>>> 2
print(next(lista))
>>> 3

... # A cada vez que um next da lista é executado, o próximo objeto é chamado pro código.
```

- Criar listas e objetos iteráveis pode se tornar algo pesado para o programa (em questão de memória), pois esses itens quando são criados, tudo que o compõe ocupa um espaço de memória, mesmo quando não utilizado. Para resolver isso, é possível usar geradores em python. Os geradores são iteráveis e iteradores ao mesmo tempo, retornando um

valor a cada vez. Ou seja, os valores não ocuparão todos os espaços de memória de uma só vez, mas sim o espaço de memória que um valor ocupa.

- Para criar um gerador, é possível fazer uma função que contenha um laço FOR juntamente com um RANGE, usando o comando *yield;*

```
import time


def gerador():
  for n in range(100):
    yield n
    time.sleep(0.2)


g = gerador()
for numero in range(100):
  print(numero) # usando o sleep fica claro o que está acontecendo na execução. Mesmo o sleep não estando no laço FOR do print, há
                # os valores, pois eles são carregaods individualmente, e não uma lista como um todo logo na execução.
```

- Outra maneira de criar um gerador, é usar algo semelhante a compreensão de listas, porém substituindo os colchetes [ ] por parênteses ( ):

```
lista = (x for x in range(100))
print(lista)

>>> <generator object <genexpr> at 0x0000020D5EA29B60> # objeto do tipo gerador que ocupa X endereço de memória
```

▼ Zip e Zip_longest - Unindo iteráveis

- É uma forma de unir listas, criando uma gerador em pares. Ou seja, se for unir por exemplo duas listas, um iterador será criado de modo que seus valores serão em pares.

```
cidades = ['São Paulo', 'Belo Horizonte', 'Salvador', 'Guarapuava']
estados = ['SP', 'MG', 'BA']
cidades_estados = zip(estados, cidade)
for conjunto in cidades_estados:
  print(conjunto)

>>> ('SP', 'São Paulo')
    ('MG', 'Belo Horizonte')
    ('BA', 'Salvador')

# perceba que a junção é feita na ordem, ou seja, indice 0 da lista 1 com indice 0 da lista 2, e assim por diante.
# A junção é feita com base na menor lista (caso haja uma menor que a outra); ou seja, perceba que a cidade 'Guarapuava' não está
# isso porq foi levado em consideração apenas a lista de estados (com 3 valores)

# Caso queira que todos os valores sejam iunidos, idependente de haver uma lista com menor quantidade de valores, é possivel usar
# da biblioteca itertools. Usando ela a saída do mesmo programa acima seria a seguinte:

>>> ('SP', 'São Paulo')
    ('MG', 'Belo Horizonte')
    ('BA', 'Salvador')
    (None, 'Guarapuava')

# Agora a cidade 'Guarapuava' está na saída do programa, acompanhada deo valor None (vazio);
# Caso queira preencher o valor None com um valor padrão ('desconhecido' por exemplo), basta adicioinar o parâmetro fillvalue e ad


# Outro detalhe que é importante, note que a junção é feita através de tuplas, ou seja, os pares são adicionados nas listas em for
```

▼ Count

- Função do biblioteca itertools que retorna um iterador;

```
from intertools import count

contador = count()
# por ser um iterador, é possível usar o next para alterar a variável
print(next(contador))
>>> 0
print(next(contador))
>>> 1 # E assim por diante
```

```
# Isso cria um contador infinito, então é preciso tomar cuidado pois é possível travar o programa/máquina de alguma forma.

# A função count pode receber os seguintes parâmetros:
  # start: indica de qual número o contador irá iniciar
  # step: indica o passo da contagem, ou seja, de quanto em quanto o contador irá pular (também funciona com número float)
    # se o step for negativo, a contagem vai ser feita de forma decrescente
```

▼ Algumas funções da biblioteca itertools

- combinations: retorna todas as possíveis combinações que os itens de uma estrutura iterável pode ter.

```
import itertools

nomes = ['Andre', 'Helena', 'Rafael']
for combinacao in itertools.combinations(nomes, 2): #sempre indicar a quantidade de grupos que deseja ser criado como combinação.
# é analisado grupo de 2 valores
  print(combinacao)

>>> ('Andre', 'Helena')
    ('Andre', 'Rafael')
    ('Helena', 'Rafael') # todas as combinações possíveis com os três nomes presentes na lista.
                         # A ordem não importa, ou seja, Andre e Helena é a mesma coisa que Helena e Andre, assim o python descart
```

- permutations: semelhante a combinations, porém agora levando em consideração a ordem dos valores.

```
import intertools

nomes = ['Andre', 'Helena', 'Rafael']
for combinacao in itertools.permutations(nomes, 2)
  print(combinacao)
>>> ('Andre', 'Helena')
    ('Andre', 'Rafael')
    ('Helena', 'Andre')
    ('Helena', 'Rafael')
    ('Rafael', 'Andre')
    ('Rafael', 'Helena')
```

- product: semelhante agora ao permutations, porém agora repetições são levadas em consideração.

```
import intertools

nomes = ['Andre', 'Helena', 'Rafael']
for combinacao in itertools.product(nomes, repet=2) # Parâmetro repet precisa ser usado aqui
  print(combinacao)

>>> ('Andre', 'Andre')
    ('Andre', 'Helena')
    ('Andre', 'Rafael')
    ('Helena', 'Andre')
    ('Helena', 'Helena')
    ('Helena', 'Rafael')
    ('Rafael', 'Andre')
    ('Rafael', 'Helena')
    ('Rafael', 'Rafael')
```

▼ Groupby - Agrupando Valores

- Função da biblioteca itertools que agrupa itens dado um determinado parâmetro. Exemplo: lista com dicionários dentro dela com algumas informações de alunos (nome e nota) e é necessário fazer um agrupamento por nota desses dados.

```
import itertools

lista = [{'nome': 'Andre', 'nota': 'A'}, {'nome': 'Helena', 'nota': 'A'}, {'nome': 'Rafa', 'nota': 'B'},
         {'nome': 'Roberto', 'nota': 'C'}, {'nome': 'Robson', 'nota': 'D'}]
ordena = lambda item: item['nota']  # aqui foi feito uma ordenação para que todas as notas ficassem em ordem
# a função groupby só funciona quando os itens estão em uma ordem;
lista.sort(key=ordena)
alunos_agrupados = itertools.groupby(lista, ordena)

for agrupamento, valores_agrupados in alunos_agrupados:  # agrupamento recebe a chave levada em consideração pra fazer
    # o agrupamento, valores_agrupados por sua vez recebe todos os dicionários presentes na lista.
    print(f'Agrupamento: {agrupamento}')
    for aluno in valores_agrupados:
```

```
        print(aluno)
    print()
```

```
from itertools import groupby, tee

alunos = [
    {'nome': 'Luiz', 'nota': 'A'},
    {'nome': 'Letícia', 'nota': 'B'},
    {'nome': 'Fabrício', 'nota': 'A'},
    {'nome': 'Rosemary', 'nota': 'C'},
    {'nome': 'Joana', 'nota': 'D'},
    {'nome': 'João', 'nota': 'A'},
    {'nome': 'Eduardo', 'nota': 'B'},
    {'nome': 'André', 'nota': 'C'},
    {'nome': 'Anderson', 'nota': 'B'},
]


def ordena(item):
  return item['nota']


alunos.sort(key=ordena)
alunos_agrupados = groupby(alunos, ordena)

'''
# Sem tee (com list)
for agrupamento, valores_agrupados in alunos_agrupados:
  valores = list(valores_agrupados)
  print(f'Agrupamento: {agrupamento}')
  for aluno in valores:
    print(f'\t{aluno}')
  quantidade = len(valores)
  print(f'\t{quantidade} alunos tiraram nota {agrupamento}')
'''

# Com tee
for agrupamento, valores_agrupados in alunos_agrupados:
  v1, v2 = tee(valores_agrupados)

  print(f'Agrupamento: {agrupamento}')

  for aluno in v1:
    print(f'\t{aluno}')

  quantidade = len(list(v2))
  print(f'\t{quantidade} alunos tiraram nota {agrupamento}')
```

▼ Map

- A função map mapeia uma função que passa por cada elemento de um iterável.

```
produtos = [
    {'nome': 'p1', 'preco': 50},
    {'nome': 'p2', 'preco': 60},
    {'nome': 'p3', 'preco': 70},
    {'nome': 'p4', 'preco': 80},
    {'nome': 'p5', 'preco': 90},
    {'nome': 'p6', 'preco': 20},
    {'nome': 'p7', 'preco': 30},
    {'nome': 'p8', 'preco': 40},
    {'nome': 'p9', 'preco': 50}
]
pessoas = [{'nome': 'Andre', 'idade': '20'}, {'nome': 'Helena', 'idade': '19'}, {'nome': 'Rafa', 'idade': '20'},
           {'nome': 'Roberto', 'idade': '58'}, {'nome': 'Robson', 'idade': '50'}]
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
from main import produtos, pessoas, lista


def aumenta(p):  # função usada para aumentar em 5% cada preço da lista produtos (linha 16)
    p['preco'] = round(p['preco'] * 1.05, 2)
    return p


# função map é semelhante a uma compreensão de listas
# entender compreensão de listas faz com que o uso da função map quase não aconteça (com listas)

lista_dobrada = map(lambda x: x * 2, lista)  # função map retorna um iterador, e não uma lista.
```

```
# o mesmo resultado seria possível com uma compreensão de listas (nesse caso, por ser uma lista de valores únicos).

print(lista)
print(list(lista_dobrada))
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #

precos = map(aumenta, produtos)  # uso da função aumenta
for preco in precos:
    print(preco)
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #

nomes = map(lambda p: p['nome'], pessoas)  # uso simples só para pegar o valor de uma chave e adicioná-lo em uma lista
# transformar em um iterável na verdade, mas sendo convertido em lista com a função list().
print(list(nomes))
```

▼ Filter

- Similar à função map, porém ela funciona filtrando os valores de uma lista ou dicionário através de uma função, retornando verdadeiro ou falso (e não um iterável). Então para a formação da nova estrutura composta, a função irá analisar o valor que a função irá retornar, caso verdadeiro, o valor continua para a nova estrutura, caso contrário, o valor não entra na nova estrutura;

```
from main import produtos, pessoas, lista # mesma estruturas usadas no tópico anterior (map).

nova_lista = filter(lambda x: x > 5, lista) # o mesmo pode ser feito com compreensão de listas
print(list(nova_lista))

novos_precos = filter(lambda p: p['preco'] > 70, produtos) # filtragem feita a partir do valor 70 (se maior que 70)
for produto in novos_precos:
    print(produto)
```

▼ Reduce

- Função da biblioteca *functools.* Essa função é um acumulador compulsivo. Reduzir não significa literalmente reduzir algum valor ou algo do tipo, na verdade significa pegar vários valores e reduzir ele até um só valor. Ou seja, vários valores → um só valor;

```
from main import produtos, pessoas, lista
from functools import reduce

soma_lista = reduce(lambda ac, i: i + ac, lista, 0)
# ac e i são respectivamente acumulador e item (nessa caso número da lista)
# i + ac é a operação que se deseja fazer com esses valores
# lista é o argumento que indica o que a função irá iterar;
# 0 indica o valor que o acumulador deve iniciar
print(soma_lista)

soma_precos = reduce(lambda ac, p: p['preco'] + ac, produtos, 0)
print(round(soma_precos, 1))
```

▼ Try, Except - Tratando exceções em Python

- Exceções são "erros" que irão parar a execução do programa;
- Para tratar esses erros é usado a estrutura try except. O try irá tentar executar determinadas linhas de códigos e, caso ocorra algum erro, o código cairá para o bloco except, que contém os códigos contento o que se deve fazer caso algum erro aconteça;
- Não é uma boa prática de programação não especificar o erro que ocorreu, ou seja, deixar somente o except. O except aceita "parâmetros" (nome das classes de erro). Por exemplo:

```
print(a) # somente isso irá levantar uma exceção
>>> NameError: name 'a' is not defined

# para tratar isso pode ser feito da seguinte maneira:

try:
  print(a)
except:
  print('Erro, não foi possível') # Esse bloco será executado e nenhuma exceção será levantada. Porém, o except não está com nenhu
                                  # então é possível e melhor fazer da seguinte maneira.

try:
  print(a)
```

```
except NameError as erro: # aqui o erro está sendo atribuído a uma variável
  print(f'Ocorreu o seguinte erro: {erro}') # e daqui em diante o código irá continuar sendo executado (caso tudo esteja correto).
```

- Mais de uma exceção pode ser tratada em um except especificando eles em uma tupla (entre parênteses e separados por vírgula);
- O bloco ELSE está presente nessa estrutura. Ele será executado somente quando o TRY for executado por completo, sem nenhuma exceção;
- Existe também o bloco FINALLY, que sempre será executado, mesmo havendo exceção ou não;

▼ Levantando exceções em Python (raise)

```
def divide(n1, n2):
    try:
        return n1 / n2
    except ZeroDivisionError as error:
        print(error)


try:
    print(divide(2, 0))
except:
    print('Erro')
```

- Fazendo somente o código acima, o comportamento do python vai ser alterado e mesmo que fora da função tente se usado um try/except, é o da função que está sendo levado em consideração. Isso pode ser contornado com o uso do `raise`;

```
def divide(n1, n2):
    try:
        return n1 / n2
    except ZeroDivisionError as error:
        print(error)
        raise


try:
    print(divide(2, 0))
except:
    print('Erro')

>>> division by zero
    Erro # o raise relançou a exceção para ser tratada em outra parte do código
```

- A própria exceção pode ser levantada usando o raise:

```
def divide(n1, n2):
    if n2 == 0:
        raise ValueError('n2 não pode ser 0.')
    return n1 / n2


print(divide(2, 0))

>>> Traceback (most recent call last):
    File "C:\cursopython\main.py", line 7, in <module>
      print(divide(2, 0))
    File "C:\cursopython\main.py", line 3, in divide
      raise ValueError('n2 não pode ser 0.')
  ValueError: n2 não pode ser 0.
```

- Página do python sobre as exceções: https://docs.python.org/3/library/exceptions.html

▼ Uso de Try e Except como condicional

```
while True:
    try:
        numero = int(input('Digite um número inteiro: '))  # caso número =! de inteiro, exceção levantada (ValueError):
    except ValueError:
        print(f'ERRO. Isso não é um número inteiro.')  # informando que houve um erro
```

```
        continue  # voltando para o início do laço para tentar novamente informar um número inteiro
    else:
        print(f'O número difitado foi {numero}')  # caso seja inteiro o bloco else é executado
        break

# Dessa forma, o try/except é usado como uma forma de condicional para informar um número inteiro.
# Se for inteiro, o programa segue, caso contrário, ele retorna ao início do laço while infinito.
```

▼ Módulos padrão do Python

- Todos os módulos padrões do python: https://docs.python.org/3/py-modindex.html
- Para usar um módulo basta digitar `import` e o nome do módulo;
- Para importar apenas uma ou mais funcionalidades específicas de um módulo, é usado `from/import` . Exemplo: `from sys import platform` ; Quando isso é feito, não é mais necessário usar o nome do módulo para chamar a função e, além disso, é possível importar dessa maneira ja nomeando a função com outro nome usando o operador `as` . Exemplo: `from sys import platform as so`

▼ Criando módulos em Python

- Em python todo arquivo de extensão `.py` é um potencial módulo. Assim, se o arquivo que é um “módulo” está na mesma pasta que o programa “main” e quiser usar a funcionalidades do módulo criado, basta importá-lo como qualquer outro módulo. `import modulo_criado` ; o uso do `from` também funciona;
- Quando um módulo é importado, todas as suas funcionalidades passam a funcionar dentro de “programa main”. Isso significa mesmo sem nenhum comando prévio, algumas linhas de código possam ser executadas;
- Usando o comando `print(__name__)` é possível ver o “nome” do arquivo. Normalmente quando executado, o retorno do print será `__main__` . Porém, se o módulo estiver sendo importado um arquivo e o mesmo ser executado (tendo o print no módulo que está sendo importado) o retorno será `__nome-modulo__` ;

▼ Criando pacotes e módulo em Python

- Em python toda pasta é um potencial pacote. O que dirá para o interpretador se aquilo é um pacote ou não é a existência de um arquivo nomeado como `__init__.py` . Esse arquivo não necessariamente precisar conter algo (pode ficar em branco). Dentro dessa pasta agora pode existir outros arquivos `.py` que poderão ser importados;
- Isso torna ajuda o programa principal a ser mais limpo e legível, além de facilitar na manutenção;
- Para importar um módulo de um pacote também é usado o `import` , da seguinte forma: `import nome-pacote.nome-modulo` ; o `from` também pode ser usado: `from nome-pacote import nome-modulo` ; e ainda existe uma última forma que é: `from nome-pacote.nome-modulo import nome-funcao` ;
- Pacotes podem receber sub-pacotes;

▼ Criando, lendo, escrevendo e apagando arquivos

- Toda a documentação sobre a função open: https://docs.python.org/pt-br/3/library/functions.html#open
- A função `open` retorna um objeto arquivo. Como parâmetro, ela irá receber o nome do arquivo e outras coisas, conforme a documentação;
- Após um arquivo ser aberto, é importante fechá-lo novamente para não ocorrer erros inesperados. Isso é feito com a função `close()` ;
- Quando um arquivo é aberto, é com se o python passasse o cursor sobre todo o conteúdo do arquivo. Porém, caso o arquivo seja aberto, modificado e tente ler ele com a função `read()` logo em seguida, nada será mostrado na tela. Isso acontece pois o cursor está logo no final do arquivo, e não tem nada para ler. Para voltar ao início do arquivo para leitura é possível usar a função `seek(arg1, arg2)` , onde normalmente é passado como parâmetro 0 e 0 (faz com que o cursor volte para a posição absoluta;
- `readline()` : lê uma linha do arquivo. A cada execução a função irá ler uma linha (cursor fica parado onde estava após a leitura na primeira chamada da função e, após ser chamada de novo o cursor passa para a próxima linha, e assim por diante);
- `readlines()` : pega cada linha do texto e joga em uma lista, onde cada índice é uma linha do texto;

- `write()`: escreve no arquivo;
- Para não precisar ficar fechando o arquivo toda vez, a maneira mais pythônica de resolver isso é usar um gerenciador de contexto chamado `with`, dessa forma, após a execução do bloco, esse gerenciador irá fechar o arquivo automaticamente. Exemplo:

```
with open('arquivo-texto.txt', 'w+' as file:
  file.write('Linha 1\n')
  file.write('Linha 2\n')
  file.write('Linha 3\n')
  file.seek(0, 0)
  print(file.read())

# Aqui onde acabou o bloco, o arquivo já irá ser fechado. Isso evita o uso de blocos try's para abrir e fechar o arquivo;
```

▼ Caminhos de módulos e pacotes

▼ Funções decoradoras e decoradores

- Controladores são objetos que estendem e/ou modificam a funcionalidade de uma função (ou método) em tempo de execução;
- Em outras palavras, o decorador funciona como uma embalagem de presente, embrulhando a função sem alterar seu conteúdo;
- Para entender decoradores é importante entender que: funções são **objetos** (uma variável pode receber uma função) e são **argumentos** (podem ser passadas para outras funções);
- Exemplo de decoradores:

```
'''
# Funções como variáveis
def fala_oi():
    print('Oi')


# A variável é igual a função
variavel = fala_oi
print(type(variavel))  # function
variavel()  # Oi
'''

'''
# Uma função dentro de outra
def master():
    # Função interna
    def slave():
        print('Oi')
    # Função a ser executada
    return slave

# Variável recebe função
variavel = master()
# Executa a função interna de master
variavel()
'''

'''
# Função como parâmetro
def master(funcao):
    # Função interna
    def slave():
        # executa a função enviada
        funcao()
    # Retorna a função interna sem executar
    return slave

# Uma função qualquer
def fala_oi():
    print('Oi')

# Variável como função
variavel = master(fala_oi)
# Executa a variável/função
variavel()
```

```
'''

'''
# Recebe uma função
def master(funcao):
    # Cria uma função interna
    def slave():
        # Decora
        print('Estou decorada.')
        # Executa a função enviada
        funcao()
    # Retorna a função interna sem executar
    return slave

# Uma função qualquer
def fala_oi():
    print('Oi')

# Decorando
fala_oi = master(fala_oi)
fala_oi()
'''

'''
# Função decoradora
def master(funcao):
    def slave():
        print('Estou decorada.')
        funcao()
    return slave

# Sintax sugar do decorador
@master
def fala_oi():
    print('Oi')

fala_oi()
'''

'''
# Decorando com parâmetros
def master(funcao):
    def slave(*args, **kwargs):
        print('Estou decorada.')
        funcao(*args, **kwargs)
    return slave

@master
def fala_oi(msg):
    print(msg)


fala_oi('Olá, sou Luiz')
'''

from time import time
from time import sleep


def velocidade(funcao):
    """
    Função decoradora: Verifica o tempo que uma função leva para executar
    """
    def envolve(*args, **kwargs):
        """ Função que envolve e executa outra função """
        # Tempo inicial
        start = time()
        # Executa a função
        resultado = funcao(*args, **kwargs)
        # Tempo final
        end = time()
        # Resultado de tempo em ms
        tempo = (end - start) * 1000
        # Mostra o tempo
        print(f'\nA função levou {tempo:.2f}ms para ser executada.')
        # Retorna a função original executada
        return resultado
    # Retorna a função que envolve
    return envolve


@velocidade
def demora(qtd):
    """ Função decorada """
    for i in range(qtd):
        print(i, end='')
```

```
# Executa a função decorada
demora(10000)
```

▼ Problema dos parâmetros mutáveis em funções

```
def lista_de_clientes(clientes_iteravel, lista=[]):
    lista.extend(clientes_iteravel)
    return lista


clientes1 = lista_de_clientes(['João', 'Maria', 'Eduardo'])
clientes2 = lista_de_clientes(['Marcos', 'Jonas', 'Zico'])
print(clientes1)
print(clientes2)

>>> ['João', 'Maria', 'Eduardo', 'Marcos', 'Jonas', 'Zico']
    ['João', 'Maria', 'Eduardo', 'Marcos', 'Jonas', 'Zico'] # as duas listas ficaram idênticas

# Para resolver o problema de passar um objeto mutável como argumento podemos fazer da seguinte forma:

def lista_de_clientes(clientes_iteravel, lista=None):
    if lista is None: # atribuir como None por padrão e depois criar a lista nova
        lista = []
    lista.extend(clientes_iteravel)
    return lista


clientes1 = lista_de_clientes(['João', 'Maria', 'Eduardo'])
clientes2 = lista_de_clientes(['Marcos', 'Jonas', 'Zico'])
print(clientes1)
print(clientes2)

>>> ['João', 'Maria', 'Eduardo']
    ['Marcos', 'Jonas', 'Zico']
```