

4 OPERADORES, VARIÁVEIS E ENTRADA E SAÍDAS

Neste capítulo iremos falar de 7 tipos de objetos, os quatro primeiros se encaixam no que chamamos de tipos de dados, sendo eles:

- Inteiros
- Floats
- Booleanos
- Strings

Os outros três são chamados de coleções, contêineres ou estruturas de dados:

- Lista
- Dicionários
- Tuplas

4.1 NÚMEROS E OPERAÇÕES ARITMÉTICAS

Quando se trata de números dentro do python conseguimos trabalhar com dois tipos de dados *int* e *float*.

4.1.1 OPERAÇÕES BÁSICAS

Vamos analisar as quatro operações básicas da matemática e os tipos de dados que elas operam:

```
[2] # Soma
    4 + 4
Python
... 8

[3] # Subtração
    4 - 3
Python
... 1

[4] # Multiplicação
    3 * 3
Python
... 9

[2] # Divisão
    3 / 2
Python
... 1.5
```

Note que nessas operações utilizamos todos os dados em formato *int*, com exceção do resultado da divisão que fora *float*, todos as respostas também foram em *int*.

Se quisermos repetir essas operações e usar os dados em *float* basta colocarmos os valores dos números em *float*.

```
[3] # Soma
    4.0 + 4.0
✓ 0.0s Python
... 8.0

[4] # Subtração
    4.0 - 3.0
✓ 0.0s Python
... 1.0

[5] # Multiplicação
    3.0 * 3.0
✓ 0.0s Python
... 9.0

[6] # Divisão
    3.0 / 2.0
✓ 0.0s Python
... 1.5
```

Note que todas as respostas resultaram em *float* também. Isso é uma das vantagens de se utilizar a linguagem interpretada, há uma tipagem automática dos valores e operações.

4.1.2 POTENCIAÇÃO

```
[6] # Potência
    4 ** 2
... 16
```

A operação de potenciação é denotada por `**` em Python. Ou seja, para calcular 4^2 , precisamos escrever `4**2`.

4.1.3 DIVISÃO COM RESULTADO EM INTEIRO

Python tem um tipo especial de divisão que é chamada de divisão inteira, em que um número é dividido por outro e depois o resultado é arredondado para baixo. O operador de divisão inteira é `//`.

Mais formalmente, quando `a` e `b` são números inteiros, `a // b` é o quociente da divisão de `a` por `b`.

Veja como a divisão inteira funciona:

```
[ ] # Divisão com resposta em int
    3 // 2
... 1.5
```

Python

4.1.4 MÓDULO OU RESTO DA DIVISÃO

Python também tem o operador `%` que calcula o resto da divisão de um número por outro.

Por exemplo, o resto da divisão de 17 por 5 é igual 2. Veja:

```
[13] 17%5
... 2
```

Python

4.1.5 COMBINANDO OPERAÇÕES

Você consegue prever o resultado das expressões abaixo?

Execute as células para verificar se você acertou!

```
[16] (8*3)//(2**2)
... 
```

Python

```
[17] 1+5*3
... 
```

Python

Neste último caso, o Python primeiro calcula o `5*3` e depois soma 1, certo?

Isso acontece porque o Python executa as operações na seguinte ordem de precedência:

1. ******
2. ***, /, // e %**
3. **+ e -**

Na dúvida, você pode colocar parênteses para deixar a ordem explícita.

Poderíamos, por exemplo, alterar a célula acima colocando ``(1+5)*3`` para indicar que o ``+`` deve ser feito antes do ``*`` .

4.1.6 FUNÇÃO TYPE

O nome da função é `type` e ela exibe o tipo de um valor ou variável. O valor ou variável, que é chamado de argumento da função, tem que vir entre parênteses. É comum se dizer que uma função ‘recebe’ um valor ou mais valores e ‘retorna’ um resultado. O resultado é chamado de valor de retorno.

```
[8] type(5)
... int

[9] type(5.0)
... float

[10] a = 'Estou estudando na Data Science Academy'
     type(a)
... str
```

4.1.7 CONVERSÃO ENTRE TIPOS DE DADOS

Podemos fazer a conversão para dados do tipo float:

```
[18] float(9)
... 9.0
```

E podemos fazer a conversão de float para int, cuidado que ele sempre irá arredondar para baixo:

```
[19] int(6.0)
... 6

[20] int(6.5)
... 6

[1] int(6.6)
... 6
```

4.1.8 FUNÇÃO ABS

É uma função que retornara o valor absoluto do argumento passado.

```
[26] # Retorna o valor absoluto
abs(-8)
... 8

[27] # Retorna o valor absoluto
abs(8)
... 8
```

4.1.9 FUNÇÃO POW

É uma função que pode ser usada para substituir a potenciação:

```
[30] # Potência
pow(5,3)
... 125
```

4.1.10 FUNÇÃO ROUND

É uma função que pode ser usada para arredondar valores matemáticos. Com o uso dessa função você poderá selecionar quantas casas decimais poderá ter no resultado, e também seguirá a regra de arredondamento padrão.

```
[3] # Retorna o valor com duas casas decimais
    round(3.14151922, 2)
✓ 0.0s Python
... 3.14

[4] # Retorna o valor com duas casas decimais
    round(3.14151922, 5)
✓ 0.0s Python
... 3.14152
```

4.2 VARIÁVEIS E ATRIBUIÇÃO

Você pode guardar um valor ou o resultado de um cálculo, ou algum conteúdo em uma variável para tornar a usá-lo.

Uma variável é composta por dois elementos:

- Nome: nome dado pelo programador à variável
- Conteúdo: valor atual da variável

Obs: no decorrer deste capítulo vamos ver que uso de variáveis é útil pois o valor/conteúdo é guardado em variável, e podemos acessar/reutilizar este valor/conteúdo várias vezes num programa, simplesmente colocando o nome de variável.

Veremos também que é possível mudar o valor de uma variável ao decorrer do programa.

Importante é saber que em cada momento, **uma variável só pode ter um único valor (e não, por exemplo, dois valores diferentes).**

Veja os exemplos abaixo:

```
# Atribuindo o valor 1 à variável var_teste
var_teste = 1
```

[3]

Python

```
# Imprimindo o valor da variável
var_teste
```

[4]

Python

... 1

```
# Não podemos utilizar uma variável que não foi definida. Leia a me
my_var
```

[5]

Python

...

```
-----
NameError                                Traceback (most recent call
/var/folders/dc/lqrc3k5j4438r150cbrdr_000000gn/T/ipykernel_1845/201204
    1 # Não podemos utilizar uma variável que não foi definida. Leia
----> 2 my_var
```

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



```
[2] var_teste = 2
✓ 0.0s Python

[3] var_teste
✓ 0.0s Python
... 2

[4] type(var_teste)
✓ 0.0s Python
... int

[5] var_teste = 9.5
✓ 0.0s Python

[6] type(var_teste)
✓ 0.0s Python
... float

[7] x = 1
✓ 0.0s Python

[8] x
✓ 0.0s Python
... 1

[9] var_teste + x
✓ 0.0s Python
... 10.5
```

4.2.1 DECLARAÇÃO MÚLTIPLAS DE VARIÁVEIS

O python possui muitos recursos que nos ajudam a trabalhar com programação de forma mais intuitiva, nos permite usar declarar diversas variáveis em uma única linha.

Vamos supor que eu precise criar três variáveis com o mesmo valor:

```
[15] fruta1 = fruta2 = fruta3 = "Melancia" Python

[16] fruta1 Python
... 'Melancia'

[17] fruta2 Python
... 'Melancia'

[18] fruta3 Python
... 'Melancia'
```

Ou precisamos criar três variáveis com valores distintos:

```
[13] pessoa1, pessoa2, pessoa3 = "Bob", "Maria", "Ana" Python

[14] pessoa1 Python
... 'Bob'

[15] pessoa2 Python
... 'Maria'

[16] pessoa3 Python
... 'Ana'
```

Tal recurso será muito útil, quando formas iterar listas e dicionários com laços de programação.

4.2.2 REGRAS PARA NOMES DE VARIÁVEIS

O nome deve seguir algumas regras ao ser criado:

Deve sempre começar com letra minúscula (o python é *case sensitive*)

Não se deve iniciar com números em hipótese alguma

Não pode haver espaços no da variável, em casos que precisar separar palavras dentro do nome da variável usar '_'.

Existem palavras reservadas do sistema que devem ser evitadas:

- | | | |
|------------|------------|----------|
| • False | • def | • if |
| • class | • from | • or |
| • finally | • nonlocal | • yield |
| • is | • while | • assert |
| • return | • and | • else |
| • None | • del | • import |
| • continue | • global | • pass |
| • for | • not | • break |
| • lambda | • with | • except |
| • try | • as | • in |
| • True | • elif | • raise |

4.2.3 BOAS PRÁTICAS PARA O NOME DAS VARIÁVEIS

Pela documentação do Python podemos usar como referencia a PEP008 (<https://peps.python.org/pep-0008/#naming-conventions>) que fala sobre os nomes dentro da programação.

Nesse material recomendamos o uso de variáveis que expressem o conteúdo que ela carrega e deva ser escrito de uma das formas:

```
# Nome em minúsculo
fruta = "Banana"

# Nome em minúsculo separado por underscore
nome_do_vendedor = "João"

# Nome com palavras capitalizadas, exceto a primeira
nomeVendedor = "Joaquim"
```

[2] ✓ 0.0s Python

4.2.4 ATRIBUIÇÃO COM OPERAÇÕES

Algumas das facilidades de programação

```
[1] # Soma com atribuição
    variavel = 0
    Python

[2] print(variavel)
    Python
... 0

[3] variavel = variavel + 1
    Python

[4] variavel += 1
    Python

[5] print(variavel)
    Python
... 2
```

Esse recurso de atribuição com operação está disponível para as seguintes operações.

```
variavel -= 1
variavel /= 3
variavel //= 2
variavel %= 2
variavel *= 3
[9] Python
```

4.3 EXERCÍCIOS – LISTA 2

Faça uma cópia do arquivo (OUT2023T – Lista 2) e comece a responder pelo notebook. Ao final, salve o arquivo em PDF conforme orientado no item 2.7.10 e envie o mesmo na atividade correspondente do Google Classroom.

4.4 MANIPULAÇÃO DE ENTRADA E SAÍDA DE DADOS

A saída de dados significa usar ferramentas que enviem ao operador do sistema um valor que foi gerado pelo código. Para essas situações nos usamos os comandos `print()` ou `display()`. (este último só funciona no Jupyter Notebook)

```
[3]: # Usando a função print para mostrar o valor do nomeVendedor
    print(nomeVendedor)
✓ 0.0s Python
... Joaquim

[4]: # Usando a função display para mostrar o valor do nomeVendedor
    display(nomeVendedor)
✓ 0.0s Python
... 'Joaquim'
```

Quando se tratar de texto para o operador podemos escrever entre aspas: como a seguir:

```
[ ]: print("Olá! Como vai?")
✓ Python
... Olá! Como vai?
```

Podemos misturar os valores de texto com variáveis, separando entre vírgula os dados:

```
[3]: print("Olá! Como vai", nomeVendedor, "?")
✓ 0.0s Python
... Olá! Como vai Joaquim ?
```

Podemos também inserir os dados usando o *format string* onde indicamos qual local do texto queremos inserir o dado da variável usando os comandos `%s(STRING)`, `%i(INT)` e `%.2f(FLOAT, permitindo arredondamento)`:

```
nomeVendedor = "Joaquim"
idade = 30
altura = 1.7365

print("Olá! Como vai %s ?" % nomeVendedor)
print("Você possui %i anos de idade." % idade)
print("Você tem %.2f metros de altura ?" % altura)
```

[9] ✓ 0.0s Python

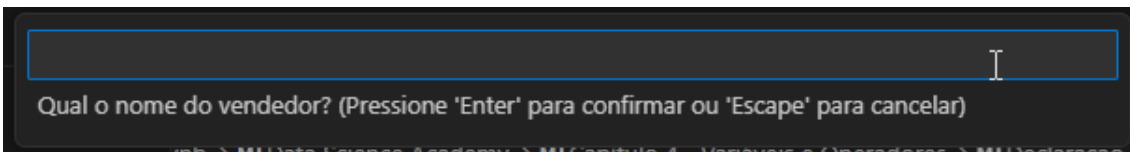
... Olá! Como vai Joaquim ?
Você possui 30 anos de idade.
Você tem 1.74 metros de altura ?

Já quando se trata de receber os dados do nosso operador para realizar os códigos precisamos usar o comando `input()`. Esse comando irá ser usado para atribuir um valor a uma variável que iremos criar. Dentro dos parênteses podemos inserir uma frase que irá ser “printada” na tela para orientar o usuário:

```
# Função input()
nomeVendedor = input("Qual o nome do vendedor? ")
```

[5] ✓ 28.9s Python

No Jupyter irá aparecer a seguinte tela para inserirmos o campo:



Se usarmos um código no arquivo .py irá aparecer direto na linha de comando.

4.5 OPERAÇÕES COM VARIÁVEIS

Operações aritméticas podem ser executadas com as variáveis normalmente, seguindo as mesmas lógicas de operações mostradas no item 4.1.1 deste documento.

```
largura = 2
```

[26] Python

[27]

```
altura = 4
```

Python

[28]

```
area = largura * altura
```

Python

[29]

```
area
```

Python

... 8

[30]

```
perimetro = 2 * largura + 2 * altura
```

Python

[31]

```
perimetro
```

Python

... 12

[34]

```
idade1 = 25
```

Python

[35]

```
idade2 = 35
```

Python

[36]

```
idade1 + idade2
```

Python

... 60

[37]

```
idade2 - idade1
```

Python

... 10

```
[38] idade2 * idade1
... 875

[39] idade2 / idade1
... 1.4

[40] idade2 % idade1
... 10
```

4.6 EXERCÍCIOS – LISTA 3

Faça uma cópia do arquivo (OUT2023T – Lista 3) e comece a responder pelo notebook. Ao final, salve o arquivo em PDF conforme orientado no item 2.7.10 e envie o mesmo na atividade correspondente do Google Classroom.

4.7 STRINGS E MANIPULAÇÃO DE STRINGS

4.7.1 DEFINIÇÕES

Vamos explicar o que é uma *string*, já utilizamos nos capítulos anteriores, mas agora iremos esclarecer e trazer novas informações. A seguir temos as definições:

- Podem ser usadas com aspas simples “, ou aspas duplas “”
- São tipos de dados que armazenam uma **lista** de caracteres.
- São imutáveis, uma vez criada ela não pode ser alterada.

```
[2] # Uma única palavra
... 'Oi'
```



```
[3] # Uma frase
    'Criando uma string em Python'
Python

... 'Criando uma string em Python'

[4] # Podemos usar aspas duplas
    "Podemos usar aspas duplas ou simples para strings em Python"
Python

... 'Podemos usar aspas duplas ou simples para strings em Python'

[5] # Você pode combinar aspas duplas e simples
    "Testando strings em 'Python'"
Python

... "Testando strings em 'Python'"
```

Para imprimir espaços em branco podemos usar o comando '\n'.

```
[7] print ('Testando \n Strings \n em \n Python')
Python

... Testando
    Strings
    em
    Python

[8] print ('\n')
Python

...
```

4.7.2 INDEXAÇÃO

Nos podemos usar indexação para controlar as strings. Lembre-se sempre que os valores de indexação ou iteração de objetos começa por 0.

```
[2] # Atribuindo uma string
    s = 'SENAI o lugar para aprender Python'
Python ✓ 0.0s

[3] print(s)
Python ✓ 0.0s

... SENAI o lugar para aprender Python
```

```
# Primeiro elemento da string
s[0]
[4] ✓ 0.0s Python
... 'S'

s[1]
[5] ✓ 0.0s Python
... 'E'

s[2]
[6] ✓ 0.0s Python
... 'N'

s[3]
[7] ✓ 0.0s Python
... 'A'

s[4]
[8] ✓ 0.0s Python
... 'I'
```

Podemos usar um ":" para executar um slicing (fatiamento) que faz a leitura de tudo até um ponto designado. Por exemplo:

```
# Retorna todos os elementos da string, começando pela posição
# (lembre-se que Python começa a indexação pela posição 0),
# até o fim da string.
s[1:]
[9] ✓ 0.0s Python
... 'ENAI o lugar para aprender Python'

# A string original permanece inalterada
s
[10] ✓ 0.0s Python
... 'SENAI o lugar para aprender Python'
```

```
[11] # Retorna tudo até a posição de índice 3
s[:3]
✓ 0.0s Python

... 'SEN'

[12] # Retorna tudo até a posição de índice 4
s[:4]
✓ 0.0s Python

... 'SENA'

[13] # Nós também podemos usar a indexação negativa e ler de trás para frente
s[-1]
✓ 0.0s Python

... 'n'

[14] # Retornar tudo, exceto a última letra
s[:-1]
✓ 0.0s Python

... 'SENAI o lugar para aprender Python'
```

Nós também podemos usar a notação de índice e fatiar a string em pedaços específicos (o padrão é 1).

Por exemplo, podemos usar dois pontos duas vezes em uma linha e, em seguida, um número que especifica a frequência para retornar elementos.

```
[15] s[::1]
✓ 0.0s Python

... 'SENAI o lugar para aprender Python'

[16] s[::2]
✓ 0.0s Python

... 'SNIoIgrpr pedrPto'

[17] s[:::-1]
✓ 0.0s Python

... 'nohtyP rednerpa arap ragul o IANES'
```

4.7.3 PROPRIEDADES DAS STRINGS

```
[27] s = "SENAI"
✓ 0.0s

# Alterando um caracter (não é possível alterar um elemento da string)
s[0] = 'x'
[22] ✗ 0.0s

-----
TypeError                                Traceback (most recent call last)
d:\996_OneDriveSenai\OneDrive - SESISENAISP - Corporativo\01_Matérias\01_FIC\03_Pr
  1 # Alterando um caracter (não é possível alterar um elemento da string)
----> 2 s[0] = 'x'

TypeError: 'str' object does not support item assignment

# Concatenando strings
s + ' é a melhor maneira de estar preparado para o mercado de trabalho!'
[28] ✓ 0.0s

'SENAI é a melhor maneira de estar preparado para o mercado de trabalho!'

s = s + ' é a melhor maneira de estar preparado para o mercado de trabalho!'
[24] ✓ 0.0s

print(s)
[25] ✓ 0.0s

SENAI é a melhor maneira de estar preparado para o mercado de trabalho!

# Podemos usar o símbolo de multiplicação para criar repetição!
letra = 'w'
[30]

letra * 3
[31]

'www'
```

4.7.4 FUNÇÕES BUILT-IN

Funções built-in, são métodos já prontos e incorporados na linguagem python. As Strings possuem diversos métodos que podemos utilizar que irão facilitar e agilizar a programação. Vamos ver a seguir alguns exemplos e como localizar.

```
[33] s = 'SENAI o lugar para aprender Python'
✓ 0.0s

[ ] s.
# capitalize
# casefold
# center
# count
# encode
# ends with
# expandtabs
# find
# format
# format_map
# index
# isalnum
s.lower()

# Upper Case
s.upper()
[36] ✓ 0.0s

... 'SENAI O LUGAR PARA APRENDER PYTHON'

# Lower case
s.lower()
[35] ✓ 0.0s

... 'senai o lugar para aprender python'

# Dividir uma string por espaços em branco (padrão)
s.split()
[37] ✓ 0.0s

... ['SENAI', 'o', 'lugar', 'para', 'aprender', 'Python']

# Dividir uma string por um elemento específico
s.split('y')
[38] ✓ 0.0s

... ['SENAI o lugar para aprender P', 'thon']
```

```
[52] ✓ 0.0s Python
... 'SENAI o lugar para aprender Python'

[40] ✓ 0.0s Python
... 'Senai o lugar para aprender python'

[53] ✓ 0.0s Python
... 4

[54] ✓ 0.0s Python
... False

[43] ✓ 0.0s Python
... False

[44] ✓ 0.0s Python
... False

[45] ✓ 0.0s Python
... False

[46] ✓ 0.0s Python
... '1000'

[47] ✓ 0.0s Python
... '1000'

[48] ✓ 0.0s Python
... str
```

4.7.5 COMPARAÇÃO DE STRINGS

```
[48] print("Python" == "R")
... False

[49] print("Python" == "Python")
... True
```

4.8 LISTAS E MANIPULAÇÃO DE LISTAS

4.8.1 DEFINIÇÃO

Antes de te mostrar como utilizar uma lista em Python, vamos começar com o que é uma lista? Nada mais é do que uma lista comum onde podemos armazenar o valor que quisermos.

Quando você faz uma lista de compras, por exemplo, isso é uma lista, e no Python é a mesma coisa. Vamos poder listar todas as informações dentro de uma única variável e depois utilizar essas informações dentro do nosso código.

Para criar listas no Python é necessário:

- Utilizar o símbolo [] (colchetes) para as listas;
- Armazenar a lista em uma VARIÁVEL;
- Separa itens da lista pela vírgula;

```
[1] # Criando uma lista
    lista_1 = ["arroz", "frango", "tomate", "leite"]
... list

[2] type(lista_1)
... list

[3] # Imprimindo a lista
    print(lista_1)
... ['arroz', 'frango', 'tomate', 'leite']
```

```
[4] # Criando outra lista
    lista_2 = ["arroz", "frango", "tomate", "leite"]
Python

[5] type(lista_2)
Python

... list

[6] # Imprimindo a lista
    print(lista_2)
Python

... ['arroz', 'frango', 'tomate', 'leite']

[8] # Criando lista
    lista_3 = [23, 100, "Programador"]
Python ✓ 0.0s

[9] type(lista_3)
Python ✓ 0.0s

... list
```

4.8.2 INDEXAÇÃO

```
[10] # Imprimindo
    print(lista_3)
Python ✓ 0.0s

... [23, 100, 'Programador']

[11] # Atribuindo cada valor da lista a uma variável.
    item1 = lista_3[0]
    item2 = lista_3[1]
    item3 = lista_3[2]
Python ✓ 0.0s

[12] # Imprimindo as variáveis
    print(item1, item2, item3)
Python ✓ 0.0s

... 23 100 Programador
```


4.8.3 EDITANDO A LISTA

Diferente das strings, nos podemos editar um dos elementos da lista, como pode ser visto na imagem a seguir:

```
[24] lista_2
✓ 0.0s Python
... ['arroz', 'frango', 'tomate', 'leite']

# Imprimindo um item da lista
[25] lista_2[2]
✓ 0.0s Python
... 'tomate'

# Atualizando um item da lista
[26] lista_2[2] = "chocolate"
✓ 0.0s Python

# Imprimindo lista alterada
[27] lista_2
✓ 0.0s Python
... ['arroz', 'frango', 'chocolate', 'leite']
```

Se quisermos deletar um elemento da lista também é possível fazer:

```
[28] lista_2
✓ 0.0s Python
... ['arroz', 'frango', 'chocolate', 'leite']

# Deletando um item específico da lista
[30] del lista_2[3]
✓ 0.0s Python

# Imprimindo o item com a lista alterada
[31] lista_2
✓ 0.0s Python
... ['arroz', 'frango', 'chocolate']
```

Se tentar deletar um item que não existe na lista teremos o seguinte erro:

```
# Não é possível deletar um item que não existe na lista.
# Vai gerar erro index out of range
del lista_2[4]
```

[29] ✖ 0.0s Python

...

```
-----
IndexError                                Traceback (most recent call
d:\996_OneDriveSenai\OneDrive - SESISENAISP - Corporativo\01_Matérias
  1 # Não é possível deletar um item que não existe na lista.
  2 # Vai gerar erro index out of range
----> 3 del lista_2[4]

IndexError: list assignment index out of range
```

4.8.4 LISTAS ANINHADAS

Listas de listas são matrizes em python, podemos realizar várias operações com esse recurso.

```
# Criando uma lista de listas
listas = [ [1,2,3], [10,15,14], [10.1,8.7,2.3] ]
```

[32] ✔ 0.0s Python

```
# Imprimindo a lista
print(listas)
```

[40] ✔ 0.0s Python

... [[1, 2, 3], [10, 15, 14], [10.1, 8.7, 2.3]]

```
# Atribuindo um item da lista a uma variável
a = listas[0][0]
```

[41] ✔ 0.0s Python

```
a
```

[42] ✔ 0.0s Python

... 1

4.8.5 OPERAÇÕES COM LISTAS

```
[ ] lista_s1 = [34, 32, 56] Python
...
[ ] lista_s1
... [34, 32, 56]

[ ] lista_s2 = [21, 90, 51] Python
...
[ ] lista_s2
... [21, 90, 51]

[ ] # Concatenando listas
    lista_total = lista_s1 + lista_s2 Python
...
[ ] lista_total
... [34, 32, 56, 21, 90, 51]
```

4.8.6 OPERADOR IN

```
[ ] # Criando uma lista
    lista_teste_op = [100, 2, -5, 3.4] Python
...
[ ] # Verificando se o valor 10 pertence a lista
    print(10 in lista_teste_op) Python
... False

[ ] # Verificando se o valor 100 pertence a lista
    print(100 in lista_teste_op) Python
... True
```

4.8.7 FUNÇÕES BUILT-IN

```
[3] # Criando uma lista
    lista_numeros = [10, 20, 50, -3.4]
    ✓ 0.0s Python

[4] # Função len() retorna o comprimento da lista
    len(lista_numeros)
    ✓ 0.0s Python
... 4

[5] # Função max() retorna o valor máximo da lista
    max(lista_numeros)
    ✓ 0.0s Python
... 50

[6] # Função min() retorna o valor mínimo da lista
    min(lista_numeros)
    ✓ 0.0s Python
... -3.4

[7] # Função sum() retorna o valor da soma de todos os elementos
    sum(lista_numeros)
    ✓ 0.0s Python
... 76.6

[ ] # Criando uma lista vazia
    a = []
    Python

[ ] print(a)
    Python
... []

[ ] type(a)
    Python
... list
```

```
[ ] a.append(10) Python

[ ] a
... [10]
```

```
[ ] a.append(50) Python

[ ] a
... [10, 50]
```

```
[ ] old_list = [1,2,5,10] Python

[ ] new_list = [] Python

[ ] # Copiando os itens de uma lista para outra
    for item in old_list:
        new_list.append(item) Python

[ ] new_list
... [1, 2, 5, 10]
```

```

[ ] cidades = ['Recife', 'Manaus', 'Salvador']
[ ] cidades.extend(['Fortaleza', 'Palmas'])
[ ] print (cidades)
Python
... ['Recife', 'Manaus', 'Salvador', 'Fortaleza', 'Palmas']

[ ] cidades.index('Salvador')
Python
... 2

Lembre-se: em Python o índice começa por 0!

[ ] cidades.index('Rio de Janeiro')
Python
...
-----
ValueError                                Traceback (most recent call
/var/folders/dc/lqrc3k5j4438r150cbrdr_000000gn/T/ipykernel_2727/38914
----> 1 cidades.index('Rio de Janeiro')

ValueError: 'Rio de Janeiro' is not in list

[ ] cidades
Python
... ['Recife', 'Manaus', 'Salvador', 'Fortaleza', 'Palmas']

[ ] cidades.insert(2, 110)
Python

[ ] cidades
Python
... ['Recife', 'Manaus', 110, 'Salvador', 'Fortaleza', 'Palmas']

[ ] # Remove um item da lista
[ ] cidades.remove(110)
Python

[ ] cidades
Python
... ['Recife', 'Manaus', 'Salvador', 'Fortaleza', 'Palmas']

```

```

# Reverte a lista
cidades.reverse()

# Imprime a lista
cidades

['Palmas', 'Fortaleza', 'Salvador', 'Manaus', 'Recife']

x = [3, 4, 2, 1]

x

[3, 4, 2, 1]

# Ordena a lista
x.sort()

x

[1, 2, 3, 4]

```

4.9 TUPLAS

4.9.1 FINALIDADE

Uma característica de uma lista em Python, é que ela é mutável. Ou seja, podemos adicionar elemento, retirar, reordenar, mudar o valor de um item etc.

Tuplas também são sequências, mas são imutáveis. Logo, uma vez criada e declarada, você não pode fazer mais nenhuma alteração nela.

Um exemplo de sequência imutável, é a string. Depois que você a declara, não pode mais alterar um caractere sequer. Até pode obter uma lista a partir dela, sub-strings etc, mas nunca vai poder alterar o valor delas na memória.

Você deve usar uma tupla sempre que tiver valores, ou itens, que nunca serão mudados, serão constantes.

Por exemplo, se você for criar um programa que use coordenadas geográficas, cada ponto da terra pode ser representado por uma tupla, e este valor jamais deverá ser alterado.

Os endereços de memória, por exemplo, também são fixos, não mudam. Seu endereço, pode ser representado por uma tupla.

Duas vantagens das tuplas em relação as listas: são mais rápidas e protegem seus scripts de sofrerem alterações. Se você armazenar sua senha em uma tupla, um hacker nunca vai conseguir mudar essa senha.

Se convencionamos que listas são para item homogêneos (tudo inteiro, tudo string etc). Já as tuplas são mais usadas para dados heterogêneos, ou seja, itens de string e número tudo junto na mesma tupla (como o nome e as notas de um aluno).

4.9.2 CONSTRUÇÃO

É definido de forma similar as listas, porém com parênteses (). A principal diferença é que tuplas são imutáveis, ou seja, ela não muda, não existe troca de elementos dentro dela.

```
[2] # Criando uma tupla
    tupla1 = ("Geografia", 23, "Elefantes", 9.8, 'Python')
Python

[3] # Imprimindo a tupla
    tupla1
Python

... ('Geografia', 23, 'Elefantes', 9.8, 'Python')
```

4.9.3 UTILIZAÇÃO

Note que não podemos usar funções que alterem o formato original da tupla, como adicionar um elemento ou excluir algum elemento, isso porque a tupla é imutável:


```
[4] # Tuplas não suportam append()
    tupla1.append("Chocolate")
Python

...
-----
AttributeError                                Traceback (most recent call
/var/folders/dc/lqrc3k5j4438r150cbrdr_000000gn/T/ipykernel_3083/25930
    1 # Tuplas não suportam append()
----> 2 tupla1.append("Chocolate")

AttributeError: 'tuple' object has no attribute 'append'
```

```
[5] # Tuplas não suportam delete de um item específico
    del tupla1["Elefantes"]
Python

...
-----
TypeError                                Traceback (most recent call
/var/folders/dc/lqrc3k5j4438r150cbrdr_000000gn/T/ipykernel_3083/36442
    1 # Tuplas não suportam delete de um item específico
----> 2 del tupla1["Elefantes"]

TypeError: 'tuple' object does not support item deletion
```

As tuplas podem ser compostas por um único elemento:

```
[6] # Tuplas podem ter um único item
    tupla1 = ("Chocolate")
Python

[7] tupla1
Python

... 'Chocolate'
```

As tuplas podem ser indexadas igual aprendemos com as listas.

```
[8] tupla1 = ("Geografia", 23, "Elefantes", 9.8, 'Python')
Python

[9] tupla1[0]
Python

... 'Geografia'
```

```
# Verificando o comprimento da tupla
len(tupla1)
```

[10]

Python

... 5

```
# Slicing, da mesma forma que se faz com listas
tupla1[1:]
```

[11]

Python

... (23, 'Elefantes', 9.8, 'Python')

```
tupla1.index('Elefantes')
```

[12]

Python

... 2

```
# Tuplas não suportam atribuição de item
tupla1[1] = 21
```

[13]

Python

...

```
-----
TypeError                                Traceback (most recent call
/var/folders/dc/lqrc3k5j4438r150cbrdr_000000gn/T/ipykernel_3083/88563
      1 # Tuplas não suportam atribuição de item
----> 2 tupla1[1] = 21
```

```
TypeError: 'tuple' object does not support item assignment
```

```
# Deletando a tupla
del tupla1
```

[14]

Python

```
tupla1
```

[15]

Python

...

```
-----
NameError                                Traceback (most recent call
/var/folders/dc/lqrc3k5j4438r150cbrdr_000000gn/T/ipykernel_3083/31414
----> 1 tupla1
```

```
NameError: name 'tupla1' is not defined
```

4.9.4 EDITANDO TUPLAS

Primeiro entenda que não é possível editar as tuplas **diretamente**. Por isso usamos os recursos de programação para contornar esse problema, vamos ver algumas formas de realizar isso.

A primeira seria usar a atribuição com operação:

```
[11] # Criando uma tupla
      t2 = ('A', 'B', 'C')
      ✓ 0.0s Python

[12] t2
      ✓ 0.0s Python
... ('A', 'B', 'C')

[13] t2 += ('D',)
      ✓ 0.0s Python

[14] t2
      ✓ 0.0s Python
... ('A', 'B', 'C', 'D')
```

Uma outra forma de fazer isso seria converter a tupla para lista, editar a lista e voltar tornar uma tupla novamente.

```
[19] # Usando a função list() para converter uma tupla para lista
      lista_t2 = list(t2)
      ✓ Python

[20] type(t2)
      ✓ Python
... tuple

[21] type(lista_t2)
      ✓ Python
... list
```

```
[22] lista_t2
Python
... ['A', 'B', 'C']

[23] lista_t2.append('D')
Python

[24] # Usando a função tuple() para converter uma lista para tupla
t2 = tuple(lista_t2)
Python

[25] t2
Python
... ('A', 'B', 'C', 'D')
```

Uma terceira forma possível seria colocar uma lista dentro da tupla, assim quando editarmos a lista, estaremos editando a tupla:

```
[7] minha_tupla = ([1,2,3], ['a', 'b', 'c'])
Python

[10] minha_tupla
Python
... ([1, 2, 3, 4], ['a', 'b', 'c'])

[9] minha_tupla[0].append(4)
Python
```

Apesar de termos essas possibilidades de editar precisamos usar esses métodos com muito cuidado, uma vez que a tupla não deveria ter a possibilidade de realizar a edição.

4.10 DICIONÁRIOS

4.10.1 FINALIDADE

Os dicionários Python são uma coleção que guarda valores multidimensionais para cada índice. Diferentemente de uma lista encadeada, que guarda apenas um valor por vez. Assim, é possível gerar estruturas mais complexas que simulam melhor a realidade e conseguem mapear instâncias do mundo real em um programa de software.

É uma coleção que consiste em chaves e valores correspondentes, quase como uma associação entre dois conjuntos. É também uma coleção ordenada, mutável, indexada e que não possibilita a inserção de elementos repetidos.

Um bom exemplo para ilustrar um dicionário é um próprio dicionário de tradução de termos de um idioma para outro. Em cada entrada, há sempre um termo correspondente, ou seja, uma chave e um valor. Para acessar um determinado valor, você pode digitar a sua chave (que funciona como um identificador) como uma forma de busca.

Um dicionário Python difere totalmente das listas, em que a pessoa programadora precisaria saber um índice numérico. Ou até mesmo das tuplas. Desse modo, torna-se uma coleção especial, adaptada para certas representações mais complexas.

A principal distinção para as listas é a questão da dimensão para cada elemento. Cada componente da coleção é composto por dois subelementos associados.

Cada um pode ser o identificador para encontrar o outro, como já pontuamos algumas vezes. Percorrer uma lista é diferente de percorrer um dicionário, assim como buscar itens também se torna diferente.

Uma das vantagens dos dicionários é a proximidade com bases de dataset que são usadas em projetos de Machine Learning e Ciência de Dados. Nesse tipo de projeto, geralmente as pessoas programadoras usam tabelas, que são matrizes de dados. Um dicionário em Python é uma ótima forma de representar uma matriz. Para isso, é preciso criar um dicionário de tuplas.

Além disso, ao aprender a lógica por trás de uma coleção mais complexa como essa, a pessoa estudante de Python facilmente se adapta a dataframes e a estruturas que são usadas em Data Science. Desse modo, o aprendizado para gerenciar algoritmos de ML e projetos desse universo se torna mais fácil e ágil.

Outro benefício é a facilidade de manipulação com esse tipo nativo do Python. A linguagem traz uma série de possibilidades para criar, editar e remover dicionários de maneira muito simples de aprender.

Outro ponto positivo é que os dicionários seguem um padrão similar ao formato JSON (JavaScript Object Notation), como o nome sugere, uma forma de notação de objetos JavaScript, e pode facilitar na integração de projetos WEB.

4.10.2 CONSTRUÇÃO

Como sempre em Python, temos uma forma simples de iniciar um dicionário. Basta fazer o uso dos { } e definir a chave e valor separados por :

```
[5] # Isso é um dicionário
    estudantes_dict = {"Pedro":24, "Ana":22, "Ronaldo":26, "Janaina":25}
Python

[6] estudantes_dict
Python
... {'Pedro': 24, 'Ana': 22, 'Ronaldo': 26, 'Janaina': 25}

[7] type(estudantes_dict)
Python
... dict

[8] estudantes_dict["Pedro"]
Python
... 24
```

4.10.3 UTILIZAÇÃO

```
[1] estudantes_dict = {"Pedro":24, "Ana":22, "Ronaldo":26, "Janaina":25} Python
✓ 0.0s

[2] estudantes_dict["Marcelo"] = 23 Python
✓ 0.0s

[3] estudantes_dict Python
✓ 0.0s
... {'Pedro': 24, 'Ana': 22, 'Ronaldo': 26, 'Janaina': 25, 'Marcelo': 23}

[4] estudantes_dict.clear() Python
✓ 0.0s

[5] estudantes_dict Python
✓ 0.0s
... {}

[10] len(estudantes_dict) Python
✓ 0.0s
... 5

[12] estudantes_dict.keys() Python
✓ 0.0s
... dict_keys(['Pedro', 'Ana', 'Ronaldo', 'Janaina', 'Marcelo'])

[15] estudantes_dict.values() Python
✓ 0.0s
... dict_values([24, 22, 26, 25, 23])
```

```
[21] estudantes_dict.items() Python
... dict_items([('Pedro', 24), ('Ana', 22), ('Ronaldo', 26), ('Janaina', 25)])

[22] estudantes2 = {"Camila":27, "Adriana":28, "Roberta":26} Python

[23] estudantes2 Python
... {'Camila': 27, 'Adriana': 28, 'Roberta': 26}

[24] estudantes_dict.update(estudantes2) Python

[25] estudantes_dict Python
... {'Pedro': 24,
    'Ana': 22,
    'Ronaldo': 26,
    'Janaina': 25,
    'Camila': 27,
    'Adriana': 28,
    'Roberta': 26}
```

É possível mesclar os tipos de estruturas entre si:

```
[50] # Dicionário de listas
dict3 = {
    'chave1':1230,
    'chave2':[22,453,73.4],
    'chave3':['picanha', 'fraldinha', 'alcatra']} Python

[51] dict3 Python
... {'chave1': 1230,
    'chave2': [22, 453, 73.4],
    'chave3': ['picanha', 'fraldinha', 'alcatra']}
```


E inclusive podemos aninhar dicionários uns dentro dos outros:

```
[58] # Criando dicionários aninhados
dict_aninhado = {'key1':{
    'key2_aninhada':{
        'key3_aninhada':'Dict aninhado em Python'
    }
}]
Python
```

```
[59] dict_aninhado
Python
... {'key1': {'key2_aninhada': {'key3_aninhada': 'Dict aninhado em P'
Python
```

```
[60] dict_aninhado['key1']['key2_aninhada']['key3_aninhada']
Python
... 'Dict aninhado em Python'
```

4.11 EXERCÍCIOS – LISTA 4

Faça uma cópia do arquivo (OUT2023T – Lista 4) e comece a responder pelo notebook. Ao final, salve o arquivo em PDF conforme orientado no item 2.7.10 e envie o mesmo na atividade correspondente do Google Classroom.