Curso: Machine Learning Básico com Python

Aula 01 - PYTHON para Inteligência Artificial

Roteiro 01 – Ambiente Colab e tipos e estruturas de dados em Python

Python é uma linguagem de propósito geral usada para as mais diversas aplicações. Atualmente Python é uma das linguagens mais utilizadas no mundo dada a sua diversidade de uso e o número enorme de bibliotecas disponíveis para as mais diversas áreas.

Uma das áreas em que Python é muito utilizado é a Inteligência Artificial (IA) principalmente no contexto da Aprendizagem de Máquina e Ciência de Dados.

Nosso objetivo aqui nesse e no próximo roteiro não será destrinchar todos os conhecimentos sobre Python, mas sim apresentar os conceitos importantes de Python que nos permita utilizar o ferramental de IA disponível principalmente no que diz respeito à manipulação tabelas e arquivos e estruturas de dados de maneira que possamos manipular o conjunto de dados que vamos analisar.

Nestes dois primeiros roteiros exercitaremos os conceitos básicos de Python sempre focado em recursos úteis para os nossos estudos de Inteligência Artificial.

Considerando a facilidade de uso, escolhemos uma versão específica de Python para trabalhar. Na verdade, um ambiente completo, repleto de bibliotecas de funções previamente instaladas que facilitarão nosso trabalho: o Google Colab.

1. COLAB

O Google Colab é um serviço de nuvem gratuito hospedado pelo Google para incentivar a pesquisa de Inteligência Artificial, que permite que você escreva e execute Python em seu navegador sem necessidade alguma de configuração, acesso gratuito à aceleração de processamento e recursos de colaboração. Além disso, tem pré-instalado uma série de bibliotecas de nosso interesse.

Nestes dois roteiros iniciais iniciais estudaremos conceitos e recursos de Python básicos.

Para entrar no Google Colab acesse o link a seguir pelo browse:

https://colab.research.google.com/

Ao clicar no link você entrará na tela a seguir com a opção “Recente” selecionada. Note que no menu superior você possui as opções conforme a figura 1.

a) Exemplos: Tem exemplos de uma série de recursos;

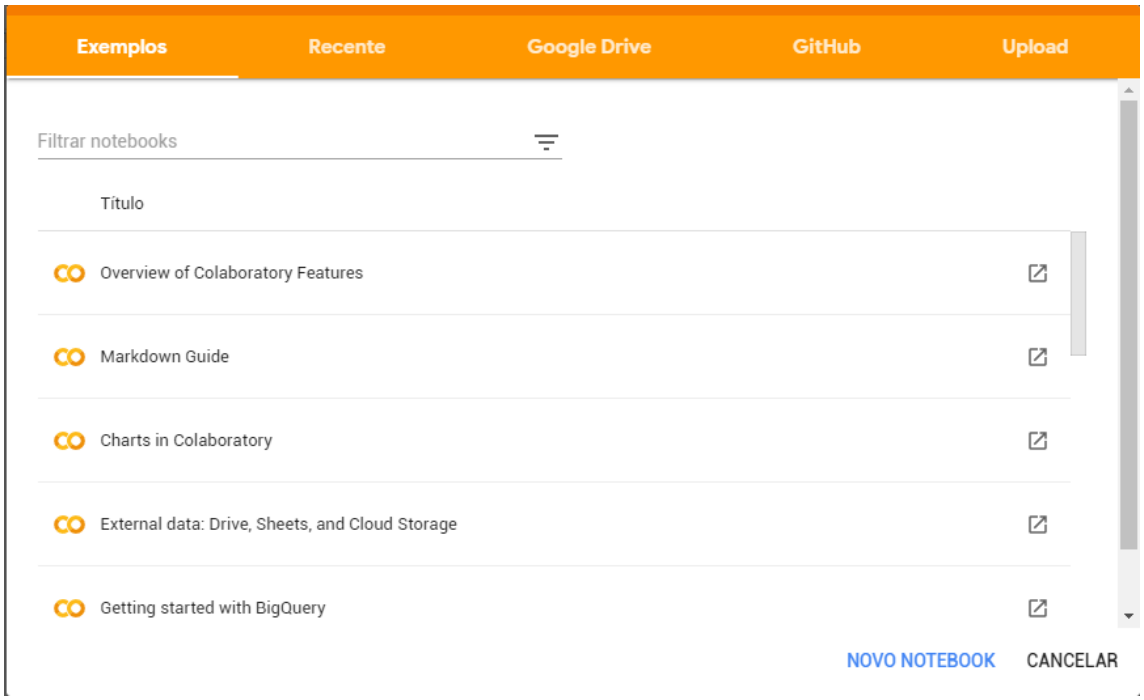
b) Recente: Permite acesso aos Notebooks abertos mais recentemente;

c) Google Drive: Acesso direto ao Notebooks do seu Google Drive quando já logado em uma conta Google;

d) GitHub: Onde você pode autentificar sua conta do GitHub para trazer Notebooks armazenados lá;

e) Upload: Permite fazer uploads de arquivos Python armazenados no seu computador.

Figura 1. Tela inicial do Google Colab

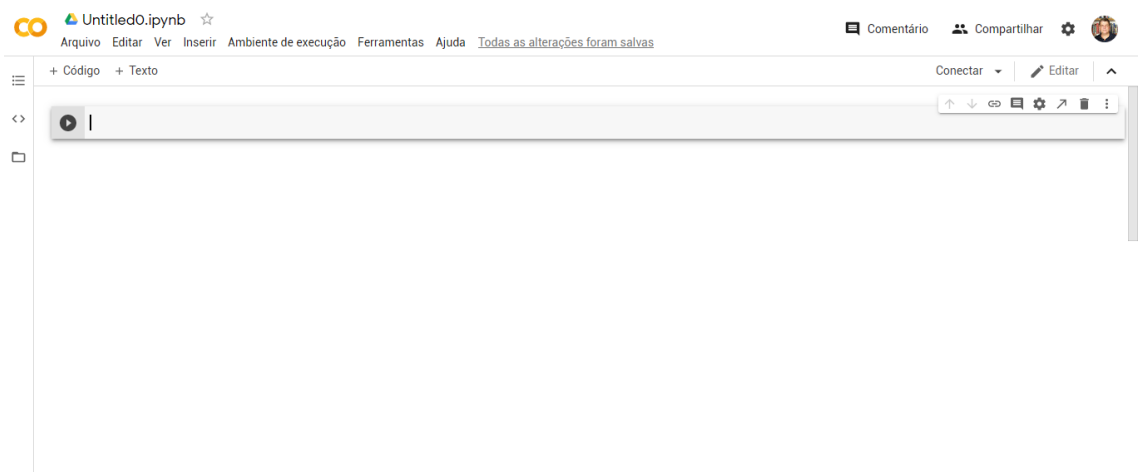


Os Notebooks Colab são Notebooks Jupyter armazenados no Google Drive que permitem combinar código executável e texto em um único documento, junto com imagens, HTML e muito mais. Quando você cria seus próprios notebooks Colab, eles são armazenados em sua conta do Google Drive. Você pode facilmente compartilhar seus notebooks Colab com seus colegas de trabalho, permitindo que eles façam comentários ou mesmo editem seus códigos.

Para criar um novo Notebook basta selecionar a opção “NOVO NOTEBOOK” da parte inferior da tela do Google Colab.

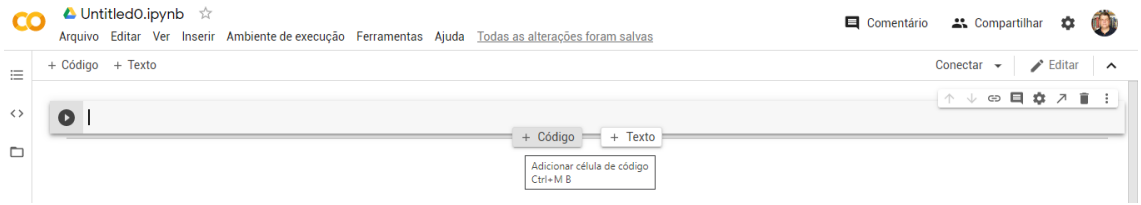
Ao criar um novo Notebook você já pode digitar seus códigos em ambiente responsivo, isto é, pode digitar um código e ver o resultado dele clicando no botão a esquerda do código digitado conforme a figura 2.

Figura 2. Tela inicial do Novo Notebook



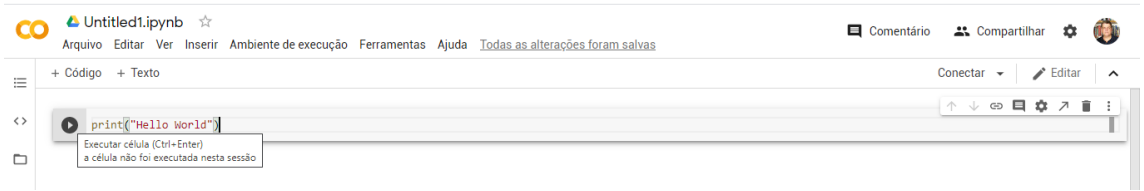
Posicionando o mouse abaixo da célula existente aparecerão opções para adicionar novas células de código ou ainda células de texto conforme figura 3. Assim você poderá ir criando e testando códigos e disponibilizando texto.

Figura 3. Opções para adicionar novas células



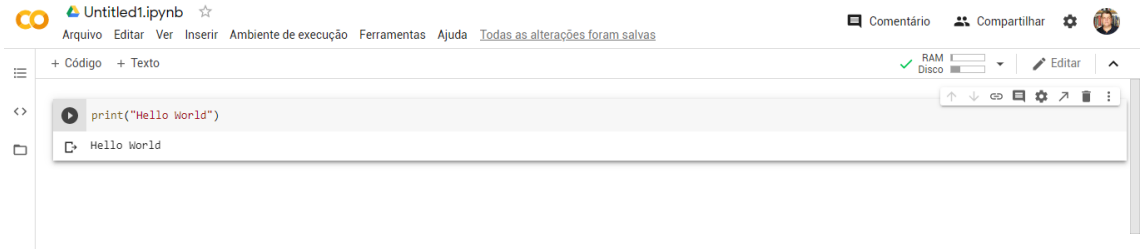
Para executar seu primeiro programa em Python, o famoso “Hello World” basta digitar o comando print (‘Hello World’) e executar clicando na seta à direita do código como na figura 4.

Figura 4. Primeiro programa



O resultado da execução aparecerá logo abaixo conforme a figura 5.

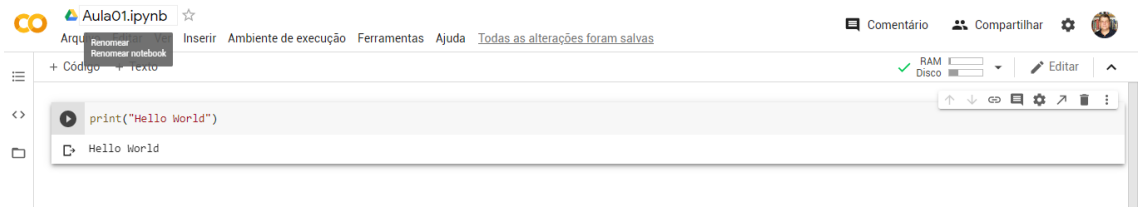
Figura 5. Resultado da execução do comando print(‘Hello world’)



E é desta forma que você irá executar seus códigos daqui em diante.

Você pode alterar o nome do seu Notebook diretamente escrevendo por sobre o nome atual do lado do símbolo do Google Drive como na figura 6 ou usar o menu de opções “Arquivo” que permite a manipulação dos arquivos de programa tanto no desktop quanto no Google Drive ou no GitHub.

Figura 6. Opção para renomear o arquivo atual



Além do menu “Arquivo” ainda existem os menus “Editar” com opções voltadas para edição dos textos, “Ver” para verificar detalhes do arquivo que está sendo editado, “Inserir” para incluir

células diversas, “Ambiente de Execução” para controles diversos relativos à execução e o menu de Ferramentas e Ajudas.

2. Tipos de Dados

A seguir serão apresentados os tipos de dados básicos e as estruturas de dados disponíveis em Python para a manipulação de dados estruturados. Os tipos de dados básicos são o numérico, lógico e conjunto de caracteres (String).

2.1. Numérico

Os quatro tipos numéricos simples, utilizados em Python, são números inteiros (int), números longos (long), números decimais (float) e números complexos (complex).

Os valores numéricos podem fazer parte de expressões aritméticas com os operadores soma (5 + 7 = 12), subtração (4 – 2 = 2), multiplicação (3 * 2 = 6), divisão (6 / 3 = 2), resto da divisão (11 % 3 = 2) e potência (5 ** 2 = 25). Podem fazer parte de expressões.

Os valores numéricos também podem fazer parte de expressões de comparação com os operadores “menor que” (a < 7), “menor ou igual” (b <= 6), “maior que” (c > 3), “maior ou igual” (d >= 5), “igual” (e == 9) e “diferente” (f != 3).

O código a seguir mostra a atribuição de valores a variáveis numéricas, as diversas formas de tratar números e os operadores de expressões com estes operadores e variáveis:

```
x = 3
print(type(x))
# Mostra "<class 'int'>"
print(x)
# Mostra "3"
print(x + 1)
# Adiciona e mostra "4"
print(x - 1)
# Subtrai e mostra "2"
print(x * 2)
# Multiplica e mostra "6"
print(x ** 2)
# Faz a exponenciação e mostra "9"
x += 1
print(x)
# Mostra "4"
x *= 2
print(x)
# Mostra "8"
y = 2.5
print(type(y))
# Mostra "<class 'float'>"
print(y, y + 1, y * 2, y ** 2)
# Mostra "2.5 3.5 5.0 6.25"
```

```
<class 'int'>
3
4
2
6
9
4
8
<class 'float'>
2.5 3.5 5.0 6.25
```

2.2. Lógico

O tipo lógico considera apenas os valores True e False. Os valores lógicos podem fazer parte de expressões lógicas com os operadores:

a) "not" onde "not a", se a é True, vira False e se é False, vira True.

b) "and" onde a expressão "(a < 3) and (c <= 7)" só será True se as duas expressões são True.

c) "or" onde a expressão "(a < 3) or (c <= 7)" só será False se as duas expressões são False.

O código a seguir mostra as diversas formas de tratar valores lógicos e os operadores de expressões com estes:

```
t = True
f = False
print(type(t))
# Mostra "<class 'bool'>"
print(t and f)
# Trabalha o operador AND e mostra "False"
print(t or f)
# Trabalha o operador OR e mostra "True"
print(not t)
# Trabalha o operador NOT e mostra "False"
print(t != f)
# Trabalha o operador XOR (ou exclusivo) e mostra "True"
```

```
<class 'bool'>
False
True
False
True
```

2.3. Strings

Strings são conjuntos de caracteres. Nele podemos guardar nomes e frases. Para considerar uma variável como string é necessário que esta receba o conjunto de caracteres entre aspas simples ou aspas duplas.

Exemplo: nome = 'Jose da Silva' ou ainda nome = "Jose da Silva"

Existem várias funções e operações que podem ser realizadas com strings como por exemplo a concatenação que é a junção entre dois strings a partir do operador soma.

Exemplo: Com n = ‘Joao’ sn = ‘Souza’ criar nome completo nc = n + sn com nome e sobrenome

Execute o código para ver o string funcionando:

```
nome = 'Jose da Silva'
print(nome)
n = 'Joao'
sn = 'Souza'
nc = n + ' ' + sn
print(nc)
```

```
Jose da Silva
Joao Souza
```

Existem uma série de funções para manipulações de strings. Segue uma relação delas:

a) len(): Retorna o tamanho da string.

Exemplo: Em teste = “Apostila de Python” o uso len(teste) retorna 18

b) capitalize(): Retorna a string com a primeira letra maiúscula.

Exemplo: Em a = "python" o uso a.capitalize() gera 'Python'

c) count(): Informa quantas vezes um caractere (ou uma sequência de caracteres) aparece na string.

Exemplo: Em b = "Linguagem Python" o uso b.count("n") retorna 2

d) startswith(): Verifica se uma string inicia com uma determinada sequência.

Exemplo: Em c = "Python" o uso c.startswith("Py") retorna True

e) endswith(): Verifica se uma string termina com uma determinada sequência.

Exemplo: Em d = "Python" o uso d.endswith("Py") retorna False

f) isalnum(): Verifica se a string possui algum conteúdo alfanumérico (letra ou número).

Exemplo: Em e = "!@#$%" o uso e.isalnum() retorna False

g) isalpha(): Verifica se a string possui apenas conteúdo alfabético.

Exemplo: Em f = "Python" o uso f.isalpha() retorna True

h) islower(): Verifica se todas as letras de uma string são minúsculas.

Exemplo: Em g = "pytHon" o uso g.islower() retorna False

i) isupper(): Verifica se todas as letras de uma string são maiúsculas.

Exemplo: Em h = "# PYTHON 12" o uso h.isupper() retorna True

j) lower(): Retorna uma cópia da string trocando todas as letras para minúsculo.

Exemplo: Em i = "#PYTHON 3" o uso i.lower() gera '#python 3'

k) upper(): Retorna uma cópia da string trocando todas as letras para maiúsculo.

Exemplo: Em j = "Python" o uso j.upper() gera 'PYTHON'

l) swapcase(): Inverte o conteúdo da string (Minúsculo / Maiúsculo).

Exemplo: Em k = "Python" o uso k.swapcase() gera 'pYTHON'

m) title(): Converte para maiúsculo todas as primeiras letras de cada palavra da string.

Exemplo: Em l = "apostila de python" o uso l.title() gera 'Apostila De Python'

n) split(): Transforma a string em uma lista, utilizando os espaços como referência.

Exemplo: Em m = "cana de açúcar" o uso m.split() gera ['cana', 'de', 'açúcar']

o) replace(S1, S2): Substitui na string o trecho S1 pelo trecho S2.

Exemplo: Em n = "Apostila teste" o uso n.replace("teste", "Python") gera 'Apostila Python'

p) find(): Retorna o índice da primeira ocorrência de um determinado caractere na string. Se o caractere não estiver na string retorna -1.

Exemplo: Em o = "Python" o uso o.find("h") retorna 3

q) ljust(): Ajusta a string para um tamanho mínimo, acrescentando espaços à direita se necessário.

Exemplo: Em p = " Python" o uso p.ljust(15) gera ' Python '

r) rjust(): Ajusta a string para um tamanho mínimo, acrescentando espaços à esquerda se necessário.

Exemplo: Em q = "Python" o uso q.rjust(15) gera ' Python'

s) center() Ajusta a string para um tamanho mínimo, acrescentando espaços à esquerda e à direita, se necessário.

Exemplo: Em r = "Python" o uso r.center(10) gera ' Python '

t) lstrip(): Remove todos os espaços em branco do lado esquerdo da string.

Exemplo: Em s = " Python " o uso s.lstrip() gera 'Python '

u) rstrip(): Remove todos os espaços em branco do lado direito da string.

Exemplo: Em t = " Python " o uso t.rstrip() gera ' Python'

v) strip(): Remove todos os espaços em branco da string.

Exemplo: Em u = " Python " o uso u.strip() gera 'Python'

Além dessas funções existe ainda a possibilidade de fatiamento de Strings que permite extrair do string apenas uma parte dos elementos de uma string.

O fatiamento é uma ferramenta usada para extrair apenas uma parte dos elementos de uma string. O formato usado é:

NomeDoString [LimiteInferior : LimiteSuperior]

Ele retorna string com os elementos das posições do limite inferior até o limite superior - 1.

Exemplos:
Para s = "Python" o uso s[1:4] seleciona os elementos das posições 1,2,3 e gera 'yth'
Para s = "Python" o uso s[2:] seleciona os elementos a partir da posição 2 e gera 'thon'
Para s = "Python" o uso s[:4] seleciona os elementos até a posição 3 e gera 'Pyth'

O código a seguir mostra o uso de algumas destas funções usadas para tratar Strings e os operadores de expressões com estes:

```
hello = 'hello'
# String literais se usa aspa simples
world = "world"
# ou aspas dupla indiferentemente
print(hello)
# Mostra "hello"
print(len(hello))
# Identifica o tamanho da String e mostra “5”
hw = hello + ' ' + world
# Operador “+” concatena Strings
print(hw)
# Mostra "hello world"
hw12 = '%s %s %d' % (hello, world, 12)
# formata como o printf da linguagem C
print(hw12)
# Mostra "hello world 12"
hw12 = '{} {} {}'.format(hello, world, 12)
print(hw12)
# Mostra "hello world 12"
s = "hello"
print(s.capitalize())
# Coloca em maiúsculo o 1a letra e mostra "Hello"
print(s.upper())
# Converte para maiúsculo tudo e mostra "HELLO"
print(s.rjust(7))
# Justifica à direita e mostra "  hello"
print(s.center(7))
# Centraliza e mostra " hello "
print(s.replace('l', '(ell)'))
# troca todos ‘l’ por ‘ell’ e mostra "he(ell)(ell)o"
print('  world '.strip())  # Tira os espaços e mostra "world"
```

```
hello
5
hello world
hello world 12
hello world 12
Hello
HELLO
  hello
```

```
 hello
he(ell)(ell)o
world
```

3. Estruturas de dados

As estruturas de dados do tipo “Containers” disponíveis são listas, dicionários, sets e tuplas.

3.1. Listas

A lista é uma estrutura mutável, ou seja, ela pode ser modificada. A seguir estão algumas funções utilizadas para manipular listas:

a) len: retorna o tamanho da lista.

Exemplo: L = [1, 2, 3, 4] e len(L) retorna 4

b) min: retorna o menor valor da lista.

Exemplo: L = [10, 40, 30, 20] e min(L) retorna 10

c) max: retorna o maior valor da lista.

Exemplo: L = [10, 40, 30, 20] e max(L) retorna 40

d) sum: retorna soma dos elementos da lista.

Exemplo: L = [10, 20, 30] e sum(L) retorna 60

e) append: adiciona um novo valor na no final da lista.

Exemplo: L = [1, 2, 3] e L.append(100) gera L [1, 2, 3, 100]

f) extend: insere uma lista no final de outra lista.

Exemplo: L = [0, 1, 2] e L.extend([3, 4, 5]) gera L [0, 1, 2, 3, 4, 5]

g) del: remove um elemento da lista, dado seu índice.

Exemplo: L = [1,2,3,4] e del L[1] gera L [1, 3, 4]

h) in: verifica se um valor pertence à lista.

Exemplo: L = [1, 2, 3, 4] e 3 in L retorna True

i) sort( ): ordena em ordem crescente

Exemplo: L = [3, 5, 2, 4, 1, 0] e L.sort( ) gera L [0, 1, 2, 3, 4, 5]

j) reverse( ): inverte os elementos de uma lista.

Exemplo: L = [0, 1, 2, 3, 4, 5] e L.reverse( ) gera L [5, 4, 3, 2, 1, 0]

Execute o código abaixo para entender melhor:

```
xs = [3, 1, 2, 'texto']
# Cria a lista
print(xs, xs[2])
# Mostra "[3, 1, 2] 2"
print(xs[-1])
```

```
# Indices negativos contam a partir do fim da lista e mostra "2"
xs[2] = 'texto'
# Listas podem ter elementos de tipos diferentes
print(xs)
# Mostra "[3, 1, 'texto']"
xs.append('outro')
# Coloca um novo elemento no fim da lista
print(xs)
# Mostra "[3, 1, 'texto', 'outro']"
x = xs.pop()
# Remove e retorna o último elemento da lista
print(x, xs)
# Mostra "bar [3, 1, 'texto']"
animais = ['gato', 'cachorro', 'macaco']
# A seguir um Loop com a lista animais
for animal in animais:
    print(animal)
# O recurso de Loop (repetição) acima conheceremos na próxima aula
```

```
[3, 1, 2, 'texto'] 2
texto
[3, 1, 'texto', 'texto']
[3, 1, 'texto', 'texto', 'outro']
bar [3, 1, 'texto', 'texto']
gato
cachorro
macaco
```

É possível também fazermos operações com diversas listas:

a) Concatenação (+): coloca a 2a lista após a1a. lista formando uma com todos elementos

Exemplo: se tivermos a = [0,1,2] e b = [3,4,5], então c = a + b gera c=[0, 1, 2, 3, 4, 5]

b) Repetição (*): cria uma lista com n vezes os elementos da original

Exemplo: se tivermos L = [1,2] então R = L * 4 gera R= [1, 2, 1, 2, 1, 2, 1, 2]

c) Fatiamento: O fatiamento de listas é semelhante ao fatiamento de strings.

Exemplo: se tivermos L = [3 , 'abacate' , 9.7 , [5 , 6 , 3] , "Python" , (3 , 'j')] então

L[1:4] seleciona os elementos das posições 1,2,3 e gera ['abacate', 9.7, [5, 6, 3]]

L[2:] seleciona os elementos a partir da posição 2 e gera [9.7, [5, 6, 3], 'Python', (3, 'j')]

L[:4] seleciona os elementos até a posição 3 e gera [3, 'abacate', 9.7, [5, 6, 3]]

Por fim, é possível criar listas com range ( ). A função range() define um intervalo de valores inteiros. Associada a list(), cria uma lista com os valores do intervalo. A função range() pode ter de 1 a 3 parâmetros (compreenderemos melhor funções na próxima aula):

- range(n) gera um intervalo de 0 a n-1

- range(i , n) gera um intervalo de i a n-1

- range(i , n, p) gera um intervalo de i a n-1 com intervalo p entre os números

Exemplos:

L1 = list(range(5)) gera [0, 1, 2, 3, 4]

L2 = list(range(3,8)) gera [3, 4, 5, 6, 7]

L3 = list(range(2,11,3)) gera [2, 5, 8]

Execute o código abaixo para entender melhor:

```
l1 = list(range(5))
print(l1)
l2 = list(range(3,8))
print(l2)
l3 = list(range(2,11,3))
print(l3)
l4 = l1 + l2
print(l4)
l5 = l3 * 4
print(l5)
l6 = [9 , 'texto' , 5.7 , [16, 11 , 18] , "Python" , [7 , 8]]
print(l6)
l7 = l6[1:4]
print(l7)
l8 = l6[2:]
print(l8)
l9 = l6[:4]
print(l9)
```

```
[0, 1, 2, 3, 4]
[3, 4, 5, 6, 7]
[2, 5, 8]
[0, 1, 2, 3, 4, 3, 4, 5, 6, 7]
[2, 5, 8, 2, 5, 8, 2, 5, 8, 2, 5, 8]
[9, 'texto', 5.7, [16, 11, 18], 'Python', [7, 8]]
['texto', 5.7, [16, 11, 18]]
[5.7, [16, 11, 18], 'Python', [7, 8]]
[9, 'texto', 5.7, [16, 11, 18]]
```

3.2. Dicionários

Um Dicionário é um conjunto de valores, um a um, associados a uma chave de acesso. Um dicionário em Python é declarado dentro de chaves com cada chave e valor separado por ":" e cada conjunto de chave e valor separados por vírgulas conforme o exemplo abaixo que tem para cada valor de produto um nome associado como chave:

Precos = {'lapis': 5.5, 'borracha': 7.0, 'caneta': 6.5}

Para mostrar todo o dicionário basta indicar o nome do dicionário no print:

Print(Precos)

Para acessar um valor específico do dicionário basta usar o nome do dicionário e a chave entre parênteses:

print("o preco da borracha eh:", Precos ['borracha'])

Algumas funções dos dicionários que podem ser úteis são:

a) del: exclui um item informando a chave.

Exemplo: del Precos[’borracha’] retirará ‘borracha’ e deixará Precos com [‘lapis’: 5.5, ‘caneta’: 6.5]

b) in: verificar se uma chave existe no dicionário.

Exemplo: ‘caneta’ in Precos retorna True e ‘caderno’ in Precos retorna False

c) keys ( ) : obtém as chaves de um dicionário.

Exemplo: Precos.keys ( ) retorna dict_keys([‘lapis’, ‘borracha’, ‘caneta’])

d) values ( ): obtém os valores de um dicionário.

Exemplo: Precos.values ( ) retorna dict_values([5.5, 7.0, 6.5])

Executando os exemplos citados temos:

```
Precos = {'lapis': 5.5, 'borracha': 7.0, 'caneta': 6.5}
print(Precos)
print("o preco da borracha eh:", Precos ['borracha'])
print(Precos.keys())
print(Precos.values())
print('caneta' in Precos)
print('caderno' in Precos)
del Precos['borracha']
print(Precos)
```

```
{'lapis': 5.5, 'borracha': 7.0, 'caneta': 6.5}
o preco da borracha eh: 7.0
dict_keys(['lapis', 'borracha', 'caneta'])
dict_values([5.5, 7.0, 6.5])
True
False
{'lapis': 5.5, 'caneta': 6.5}
```

A seguir mais um exemplo para executar:

```
d = {'gato': 'cat', 'cachorro': 'dog'}
# Cria um novo dicionário com alguns dados
print(d['gato'])
# Identifica significado de uma entrada e mostra "cat"
print('gato' in d)
# Verifica se uma chave está e mostra "True"
print(d)
d['peixe'] = 'fish'
# Inclui um element em um dicionário
```

```
print(d['peixe'])
# Mostra "fish"
print(d['macaco'])
# KeyError: 'macaco' porque não é uma chave do dicionário d
print(d.get('macaco', 'N/A'))
# Busca um elemento com valor default para falha, mostra "N/A"
print(d.get('peixe', 'N/A'))
# Busca um elemento com valor default para falha, mostra "fish"
del d['peixe']
# Remove um element do dicionário
print(d.get('peixe', 'N/A'))
# Busca um elemento com valor default para falha, mostra "N/A"
```

```
cat
True
{'gato': 'cat', 'cachorro': 'dog'}
fish
N/A
fish
N/A
```

3.3. Sets

Sets são conjuntos não ordenados de elementos. Para sets usamos chaves.

```
animais = {'gato', 'cachorro'}
print('gato' in animais)
# Verifica se um elemento está no set e mostra "True"
print('peixe' in animais)
# Mostra "False"
animais.add('peixe')
# Inclui um elemento no set
print('peixe' in animais)
# Mostra "True"
print(len(animais))
# Identifica o número de elementos do set e mostra "3"
animais.add('gato')
# Incluir um elemento que já existem não faz nada
print(len(animais))
# Mostra "3"
animais.remove('gato')
# Remove um elemento do set
print(len(animais))
# Mostra "2"
```

```
True
False
True
3
```

```
3
2
```

3.4. Tuplas

Tuplas são listas que não podem ser alteradas. Em tuplas usamos parênteses “( )” ao invés de colchetes “[ ]” como em listas.

```
tp1 = (1,2,3,4,5,6)
# Cria uma tupla
print(tp1)
# Mostra a tupla (1, 2, 3, 4, 5, 6)
print(tp1[4])
# Mostra “5”
```

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

Um segundo exemplo usando dicionários com tuplas:

```
d = {(0, 1): 0, (1, 2): 1, (2, 3): 2, (3, 4): 3, (4, 5): 4,
 (5, 6): 5, (6, 7): 6, (7, 8): 7, (8, 9): 8, (9, 10): 9}
# Cria dicionários com tuplas como chaves
print(d)
# Mostra todas as tuplas do dicionário
tp2 = (5, 6)
# Cria uma tupla
print(type(tp2))
# Mostra "<class 'tuple'>"
print(d[tp2])
# Mostra "5"
print(d[(1, 2)])
# Mostra "1"
```

```
{(0, 1): 0, (1, 2): 1, (2, 3): 2, (3, 4): 3, (4, 5): 4, (5, 6): 5,
(6, 7): 6, (7, 8): 7, (8, 9): 8, (9, 10): 9}
<class 'tuple'>
5
1
```

Uma forma alternativa de fazer a construção de tuplas usando iterações (um recurso que conheceremos na próxima aula):

```
d = {(x, x + 1): x for x in range(10)}
# Cria dicionários com tuplas como chaves
print(d)
# Mostra todas as tuplas do dicionário
# {(0, 1): 0, (1, 2): 1, (2, 3): 2, (3, 4): 3, (4, 5): 4,
# (5, 6): 5, (6, 7): 6, (7, 8): 7, (8, 9): 8, (9, 10): 9}
tp2 = (5, 6)
# Cria uma tupla
```

```
print(type(tp2))
# Mostra "<class 'tuple'>"
print(d[tp2])
# Mostra "5"
print(d[(1, 2)])
# Mostra "1"
```

```
{(0, 1): 0, (1, 2): 1, (2, 3): 2, (3, 4): 3, (4, 5): 4, (5, 6): 5,
(6, 7): 6, (7, 8): 7, (8, 9): 8, (9, 10): 9}
<class 'tuple'>
5
1
```