



Aula 1
Introdução ao
Python - Parte I

Motivação

- Python é uma das linguagens de programação de maior crescimento nos últimos anos
- Em vários rankings de “uso de linguagens de programação” e “quais linguagens você deve aprender”, Python aparece frequentemente entre as 3 principais linguagens, juntamente com Java e JavaScript
- As principais bibliotecas de Aprendizado de Máquina e Inteligência Artificial estão sendo escritas em Python nos dias atuais
- Além disso, o Python compõe várias funcionalidades de aplicativos como o PostgreSQL e o OpenOffice

Motivação

- **Razões:**
 - Codificação mais simples que outras linguagens
 - Fácil manipulação de dados
 - Estruturas de dados eficientes e com facilidades de operações matemáticas
- Página oficial: <https://www.python.org/>

História

- O Python foi concebido no final de 1989 por Guido van Rossum no Instituto de Pesquisa Nacional para Matemática e Ciência da Computação (CWI)
- Versões:
 - 1.0: 1994
 - 2.0: 2000
 - 3.0: 2008
 - 3.7.4: julho de 2019
- Atualmente, Python é um dos componentes padrão de vários sistemas operacionais, entre eles estão a maioria das distribuições do Linux, AmigaOS 4, FreeBSD, NetBSD, OpenBSD e OS X.
- O Python possui um modelo de desenvolvimento comunitário, aberto e gerenciado pela organização sem fins lucrativos Python Software Foundation

Características

- Python é uma linguagem de programação:
 - Alto nível
 - Interpretada
 - Script
 - Multiparadigma:
 - Programação procedural
 - Programação imperativa
 - Orientação a objetos
 - Programação funcional
 - Tipagem dinâmica

Características: alto nível

- Uma linguagem com um nível de abstração relativamente elevado
- Longe do código de máquina e mais próximo à linguagem humana

```
nome = input('Digite o nome:')  
if nome == 'Rafael': print("Fiu fiu!")
```

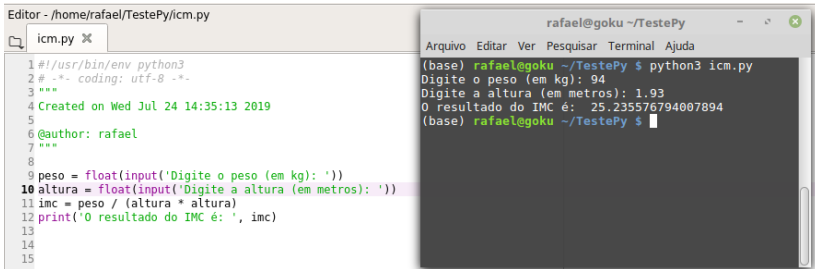
```
nota = 15  
if 0 <= nota <= 10:  
    print('Nota Válida')  
else:  
    print('Nota Inválida')
```

Nota Inválida

Características: linguagem interpretada

- O código fonte é executado por um programa de computador chamado interpretador, que em seguida é executado pelo sistema operacional
- Mesmo que um código em uma linguagem passe pelo processo de compilação, a linguagem pode ser considerada interpretada se o programa resultante não for executado diretamente pelo sistema operacional ou processador (caso dos *bytecodes* do Java)
- Isso possibilita o paradigma “*Write Once, Run Everywhere (WORA)*”
- **OBSERVAÇÃO:** é possível compilar códigos Python em linguagem de máquina ou em *bytecodes* de outras linguagens, como o Java

Características: linguagem interpretada



The image shows a code editor window on the left and a terminal window on the right. The code editor displays a Python script named `icm.py` with the following content:

```
1#!/usr/bin/env python3
2# -*- coding: utf-8 -*-
3"""
4Created on Wed Jul 24 14:35:13 2019
5
6@author: raphael
7"""
8
9peso = float(input('Digite o peso (em kg): '))
10altura = float(input('Digite a altura (em metros): '))
11imc = peso / (altura * altura)
12print('O resultado do IMC é: ', imc)
13
14
15
```

The terminal window, titled `rafael@goku ~/TestePy`, shows the execution of the script:

```
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
(base) raphael@goku ~/TestePy $ python3 icm.py
Digite o peso (em kg): 94
Digite a altura (em metros): 1.93
O resultado do IMC é: 25.235576794007894
(base) raphael@goku ~/TestePy $
```


Características: programação imperativa

- Programação imperativa é um paradigma de programação que descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa (também conhecida como programação procedural)
- 4 componentes fundamentais:
 - Variáveis: modelam as células de memória
 - Comandos de atribuição: são baseados nas operações de transferências de dados e instruções.
 - Execução sequencial de procedimentos
 - Forma iterativa de repetição

Características: programação imperativa

```
tabuada = 2
contador = 1
while contador <= 10:
    resultado = contador * tabuada
    print(tabuada, 'x', contador, '=', resultado)
    contador = contador + 1
```

```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

Características: programação procedural

- Paradigma de programação baseado no conceito de chamadas de procedimentos
- Utilizado para modularizar procedimentos e evitar redundâncias de código

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

```
soma(10,15)
```

25

Características: programação orientada a objetos

- Toda variável em Python é um objeto, i.e., pode conter atributos e métodos originados a partir de uma classe

```
nome = 'Rafael Rossi'
```

```
type(nome)
```

```
str
```

```
nome.
```

- capitalize
- casefold
- center
- count
- encode
- endswith
- expandtabs
- find
- format
- format_map

Características: programação funcional

- Programação funcional é um paradigma de programação que trata a computação como uma avaliação de funções
- Permite a atribuição de funções a variáveis e passagens de funções como argumentos de outras funções
- Permite sobrescrever operadores para realizar chamadas de funções

Características: programação funcional

```
[3]: def soma(a,b):  
      return a + b
```

```
[4]: op = soma
```

```
[7]: def operacao(funcao, a, b):  
      return funcao(a,b)
```

```
[8]: operacao(op,5,10)
```

```
[8]: 15
```

Características: tipagem dinâmica

- Tipagem dinâmica significa alterar o tipo de uma variável dinamicamente e de maneira implícita apenas baseando-se no valor atribuído

```
In [1]: variavel = 10
```

```
In [2]: type(variavel)
```

```
Out[2]: int
```

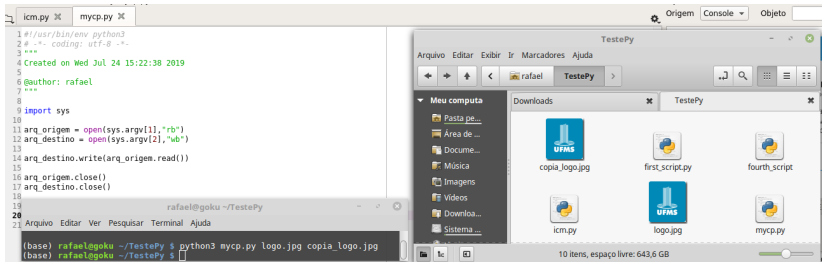
```
In [3]: variavel = 'Professor lindão'
```

```
In [4]: type(variavel)
```

```
Out[4]: str
```

Características: linguagem de *script*

- O termo “linguagem de *script*” é também usado para se referir à linguagens de propósitos diversos, ou ainda se referir à linguagens que permitem fazer pequenos programas de forma rápida



Como programar

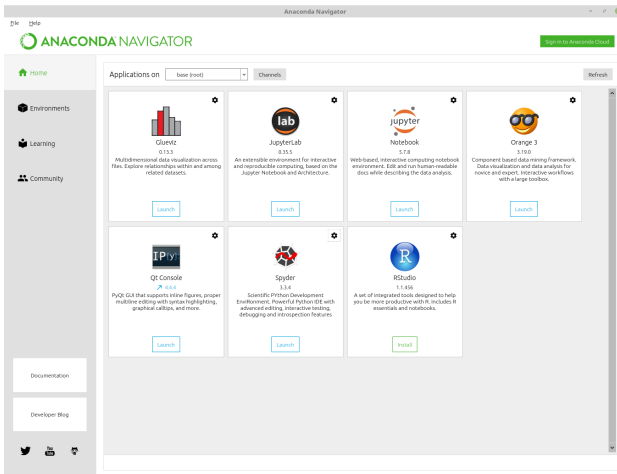
- Existem diversas formas de se programar e executar um código em Python
 - *Console (Shell ou Terminal)* interativo
 - Integrated Development Environment (IDE)
 - *Notebook*
- Pode-se utilizar o *framework* ANACONDA para se programar em Python com todos os ambientes mencionados acima



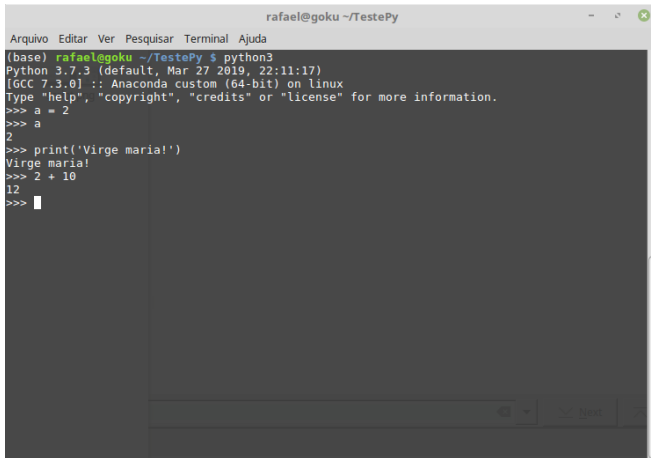
<https://www.anaconda.com/products/individual>

Como programar

ANACONDA Navigator (no Linux)

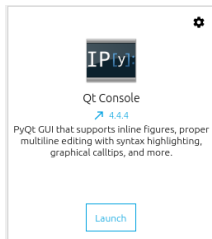


Programando no Console



```
rafael@goku ~/TestePy
Arquivo Editar Ver Pesquisar Terminal Ajuda
(base) rafaelf@goku ~/TestePy $ python3
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0] :: Anaconda custom (64-bit) on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 2
>>> a
2
>>> print('Virge maria!')
Virge maria!
>>> 2 + 10
12
>>> █
```

Programando no Console



```
Jupyter QtConsole
File Edit View Kernel Window Help
Jupyter QtConsole 4.4.4
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: a=2
In [2]: a
Out[2]: 2

In [3]: print('Virge')
Virge

In [4]: 2*10
Out[4]: 20

In [5]: nome='Rafael'

In [6]: nome.
capitalize format isidentifier ljust rjust swapcase
casefold format_map islower lower rpartition title
center index isnumeric lstrip rsplit translate
count isalnum isprintable maketrans rstrip upper
encode isalpha isspace partition split zfill
endswith isascii istitle replace splitlines
expandtabs isdecimal isupper rfind startswith
find isdigit join rindex strip
```

OBSERVAÇÃO: para sair do console utilize a função `quit()`

Programando em uma IDE

Spyder (Python 3.7)

Arquivo Editar Pesquisar Código Executar Depurar Consoles Projetos Ferramentas Ver Ajuda

Editor - /home/rafael/TestePy/mycp.py

```
1#!/usr/bin/env python3
2# -*- coding: utf-8 -*-
3"""
4Created on Wed Jul 24 15:22:38 2019
5
6@author: rafael
7"""
8
9import sys
10
11arq_origem = open(sys.argv[1],"rb")
12arq_destino = open(sys.argv[2],"wb")
13
14arq_destino.write(arq_origem.read())
15
16arq_origem.close()
17arq_destino.close()
18
19
20|
```

Explorador de variáveis

Nome	Tipo	Tamanho	Valor
altura	float	1	1.71
imc	float	1	84.0
peso	float	1	84.0

Ajuda Explorador de variáveis Explorador de arquivos

Console IPython

Console 3/A

Python 3.7.3 (default, Mar 27 2019, 22:11:17)
Type "copyright", "credits" or "license" for more information.

IPython 7.5.0 -- An enhanced Interactive Python.

In [1]: runfile('/home/rafael/TestePy/icm.py', wdir='/home/rafael/TestePy')

Digite o peso: 84

Digite a altura: 1.71

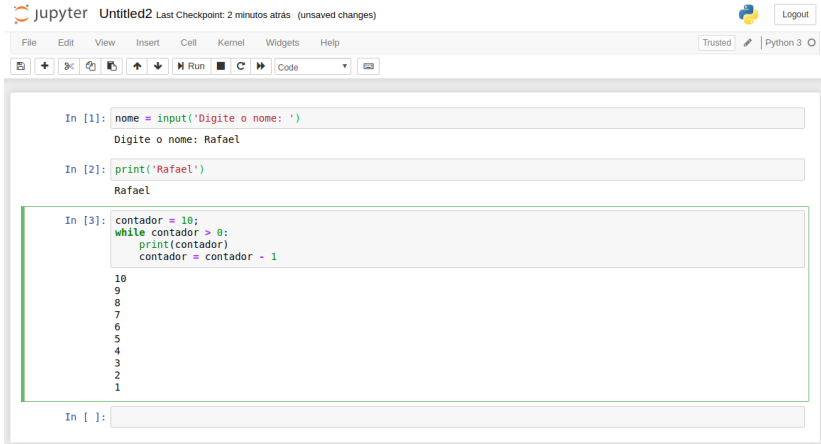
O resultado do IMC é: 84.0

In [2]:

Console IPython Log do histórico

Permissões: RW Fim de linha: LF Codificação: UTF-8 Linha: 20 Coluna: 1 Memória: 87 %

Programando em um *Notebook*



jupyter Untitled2 Last Checkpoint: 2 minutos atrás (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
In [1]: nome = input('Digite o nome: ')
        Digite o nome: Rafael

In [2]: print('Rafael')
        Rafael

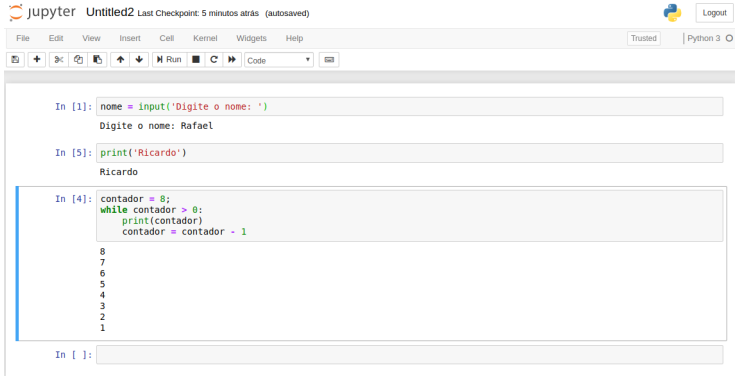
In [3]: contador = 10;
        while contador > 0:
            print(contador)
            contador = contador - 1


        10
        9
        8
        7
        6
        5
        4
        3
        2
        1

In [ ]:
```

Programando em um *Notebook*

- Um notebook permite rapidamente re-editar um trecho de código e executá-lo novamente



jupyter Untitled2 Last Checkpoint: 5 minutos atrás (autosaved)  Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

In [1]: `nome = input('Digite o nome: ')`
Digite o nome: Rafael

In [5]: `print('Ricardo')`
Ricardo

In [4]: `contador = 8;`
`while contador > 0:`
 `print(contador)`
 `contador = contador - 1`
8
7
6
5
4
3
2
1

In []:

Opções *On-line*

- Vale ressaltar que existem opções on-line que disponibilizam terminais ou *notebooks*
 - Terminais:
 - <https://www.python.org/shell/>
 - <https://repl.it/languages/python3>
 - *Notebooks*:
 - <https://jupyter.org/try>
 - <https://cocalc.com/doc/jupyter-notebook.html>
 - <https://colab.research.google.com/>

Criando Variáveis

- Tipos básicos para armazenar valores únicos
 - Inteiro (int)
 - Ponto flutuante (float)
 - Booleano (bool)
 - Complexo (complex)
- **OBSERVAÇÃO:** até os tipos básicos são objetos

Criando Variáveis

```
In [1]: var1 = 2
In [2]: var1
Out[2]: 2

In [3]: type(var1)
Out[3]: int

In [4]: var2 = 2.1
In [5]: var2
Out[5]: 2.1

In [6]: type(var2)
Out[6]: float

In [7]: var3 = 'Rafael'
In [8]: var3
Out[8]: 'Rafael'

In [9]: type(var3)
Out[9]: str

In [11]: var4 = True
In [12]: var4
Out[12]: True

In [13]: type(var4)
Out[13]: bool

In [14]: var5 = 4.5j
In [16]: var5
Out[16]: 4.5j

In [17]: type(var5)
Out[17]: complex
```

Criando Variáveis

- Em Python, por convenção, se utiliza o padrão *snake case* para definir nomes de variáveis e funções
- O padrão *snake case* consiste em separar cada nome que compõem a variável com *underlines* “_”

```
In [7]: nome_do_caboclo = 'Rafael'  
        idade_do_peao = 34
```

Strings

- *Strings* podem ser criadas com aspas simples ou duplas

```
In [1]: nome1 = "Rafael"
```

```
In [2]: nome1
```

```
Out[2]: 'Rafael'
```

```
In [4]: nome2 = 'Rafael'
```

```
In [5]: nome2
```

```
Out[5]: 'Rafael'
```

- Quando utilizar aspas simples, é possível utilizar aspas duplas sem sequência de escape
- O contrário também é válido

```
In [4]: frase_capitao_nascimento = 'O Capitão Nascimento disse: "É você quem financia essa merda aqui!"'
```

```
In [5]: frase_capitao_nascimento
```

```
Out[5]: 'O Capitão Nascimento disse: "É você quem financia essa merda aqui!"'
```

Criando Variáveis

- Tipos básicos para armazenar conjuntos de valores:
 - **list** (lista)- para agrupar um conjunto de elementos
 - *tuple* (tupla) - semelhante ao tipo `list`, porém, imutável
 - **dic** (dicionário) - para agrupar elementos que serão recuperados por uma chave
 - **set** (conjunto) - armazenar conjuntos de valores

Listas

- As listas são utilizadas normalmente para armazenar elementos de um único tipo
- Entretanto, o Python permitir armazenar objetos de diferentes tipos
- Operações em listas são bem simples
 - Criação com o uso de `[]` e com os elementos separados por vírgula
 - Concatenação de dois elementos com o símbolo operador de adição
- As listas possuem métodos para a adição, remoção, busca, ...
- Elementos da lista podem ser acessado pelo índice utilizando a notação de acesso a posições de vetor → `lista[índice]`

Listas

```
In [1]: numeros = [1,3,5,7,9,11] #criando uma lista

In [2]: numeros
Out[2]: [1, 3, 5, 7, 9, 11]

In [3]: numeros2 = [1,3,5] + [7,9,22] #concatenando duas listas

In [4]: numeros2
Out[4]: [1, 3, 5, 7, 9, 22]

In [5]: numeros2.append(1) #adicionando um único elemento

In [6]: numeros2
Out[6]: [1, 3, 5, 7, 9, 22, 1]

In [7]: numeros2.index(7) #retornando o índice de um elemento na lista
Out[7]: 3

In [8]: numeros2[1] #acessando um elemento da lista
Out[8]: 3

In [10]: """removendo um elemento da lista. O argumento da função
remove() e o elemento a ser removido, não o índice"""
numeros2.remove(7)

In [11]: numeros2
Out[11]: [1, 3, 5, 9, 22, 1]
```

Listas

- Listas podem ser utilizadas para armazenar dados de diferentes tipos

```
In [9]: lista = [1, 2, 3, 'São Paulo', 'Rio de Janeiro']
```

```
In [10]: lista
```

```
Out[10]: [1, 2, 3, 'São Paulo', 'Rio de Janeiro']
```


Listas

- Podemos também fazer listas de listas

```
In [1]: #cada elemento da lista é uma lista  
lista_de_listas = [[1,2,3], [4,5,6], [7,8,9]]
```

```
In [2]: lista_de_listas
```

```
Out[2]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
In [3]: lista_de_listas[0]
```

```
Out[3]: [1, 2, 3]
```

```
In [4]: lista_de_listas[0][1]
```

```
Out[4]: 2
```

```
In [6]: #cada elemento da lista pode ter tamanho variado  
lista_de_listas.append([10,11,12,13])
```

```
In [7]: lista_de_listas
```

```
Out[7]: [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12, 13]]
```

```
In [8]: #concatenando utilizando o +  
lista_de_listas += [[14,15,16]]
```

```
In [9]: lista_de_listas
```

```
Out[9]: [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12, 13], [14, 15, 16]]
```

Listas

- Operações de *slicing*: retornam uma subsequência da sequência original

```
In [1]: #Criando uma linha para testar os slicings  
numeros = [1,3,5,7,9,11,13,15,17,16]
```

```
In [2]: #Especificando um índice inicial e um final  
numeros[2:6] #O último índice é não inclusivo
```

```
Out[2]: [5, 7, 9, 11]
```

```
In [3]: #Se especificarmos só o índice final, o índice inicial é 0  
numeros[:6]
```

```
Out[3]: [1, 3, 5, 7, 9, 11]
```

```
In [4]: """Se especificarmos só o índice inicial, o índice final  
corresponde ao tamanho da lista"""  
numeros[6:]
```

```
Out[4]: [13, 15, 17, 16]
```

```
In [5]: """Se por um acaso quiser retornar a lista toda utilizando slicing"""  
numeros[0:len(numeros)] #A função len() retorna o tamanho da lista
```

```
Out[5]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 16]
```

```
In [6]: """Já que se omitir o inicial equivale a 0, e omitir o final  
equivale a len([lista]), o mesmo efeito acima pode ser  
obtido por:"""  
numeros[:]
```

```
Out[6]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 16]
```

Listas

- *Slicing* com valores negativos

```
In [9]: numeros
```

```
Out[9]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 16]
```

```
In [10]: numeros[:-1]
```

```
Out[10]: [1, 3, 5, 7, 9, 11, 13, 15, 17]
```

```
In [11]: numeros[-2:]
```

```
Out[11]: [17, 16]
```

```
In [12]: numeros[-5:-3]
```

```
Out[12]: [11, 13]
```

Listas

- *Slicing* com passos

```
In [1]: numeros = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

```
In [4]: numeros[::2] #0 último índice corresponde ao tamanho do passo
```

```
Out[4]: [1, 5, 9, 13, 17]
```

```
In [5]: numeros[::3]
```

```
Out[5]: [1, 7, 13, 19]
```

```
In [6]: numeros[::-2] #Dá para usar tamanhos de passos negativos
```

```
Out[6]: [19, 15, 11, 7, 3]
```

```
In [7]: numeros[::-1] #Dá para inverter a lista com passo negativo
```

```
Out[7]: [19, 17, 15, 13, 11, 9, 7, 5, 3, 1]
```

Listas

● Manipulando elementos da lista com *slicing*

```
In [36]: numeros = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
          numeros2 = numeros.copy()
          numeros3 = numeros.copy()
          numeros4 = numeros.copy()

In [27]: #substituindo os elementos de índices 0 a 3 pelo valor 0
          numeros2[0:4] = [0]

In [28]: numeros2
Out[28]: [0, 9, 11, 13, 15, 17, 19]

In [30]: numeros3[0:4] = [] #eliminando os elementos de índices 0 a 3

In [31]: numeros3
Out[31]: [9, 11, 13, 15, 17, 19]

In [32]: numeros4[:] = [] #limpando a lista => equivalente à função empty()

In [33]: numeros4
Out[33]: []

In [39]: numeros[2:3] = [] #eliminando um único elemento da lista pelo índice

In [40]: numeros
Out[40]: [1, 3, 7, 9, 11, 13, 15, 17, 19]

In [41]: numeros[::2] = [5,5,5,5,5] #Substituindo elementos com passos#

In [42]: numeros
Out[42]: [5, 3, 5, 9, 5, 13, 5, 17, 5]
```

Tuplas

- Tuplas são tipicamente utilizadas para armazenar dados heterogêneos
- Tuplas são criadas utilizando "(" e ")" e cada elemento da tupla deve ser separado por ","
- Tuplas são imutáveis, i.e., o conteúdo de um elemento não pode ser alterado após criado
- Cada elemento da tupla pode ser um objeto mutável ou não mutável
- Para acessar um elemento de uma tupla, a notação é a mesma da lista → `tupla[índice]`

Tuplas

```
In [45]: tupla1 = ('Rafael', 'Professor', 'XXX.XXX.XXX-XX') #Criando uma tupla
```

```
In [46]: tupla1
```

```
Out[46]: ('Rafael', 'Professor', 'XXX.XXX.XXX-XX')
```

```
In [47]: tupla2 = ('Marcos Paulo', 'Miliciano', "ZZZ.ZZZ.ZZZ-ZZ")
```

```
In [48]: tupla2
```

```
Out[48]: ('Marcos Paulo', 'Miliciano', 'ZZZ.ZZZ.ZZZ-ZZ')
```

```
In [49]: #Acessar os elementos da tupla é semelhante a acessar os elementos de uma lista  
tupla2[2]
```

```
Out[49]: 'ZZZ.ZZZ.ZZZ-ZZ'
```

```
In [50]: lista_tupla = [tupla1, tupla2] #Criando uma lista de tuplas
```

```
In [51]: lista_tupla
```

```
Out[51]: [('Rafael', 'Professor', 'XXX.XXX.XXX-XX'),  
          ('Marcos Paulo', 'Miliciano', 'ZZZ.ZZZ.ZZZ-ZZ')]
```

Tuplas

```
In [52]: tupla1[0] = 'Ricardo' #Tuplas são imutáveis
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-52-94925c6b0207> in <module>  
----> 1 tupla1[0] = 'Ricardo'  
  
TypeError: 'tuple' object does not support item assignment
```

```
In [53]: tupla1 += tupla2 #apesar de imutáveis, é possível concatenar tuplas
```

```
In [54]: tupla1
```

```
Out[54]: ('Rafael',  
          'Professor',  
          'XXX.XXX.XXX-XX',  
          'Marcos Paulo',  
          'Miliciano',  
          'ZZZ.ZZZ.ZZZ-ZZ')
```


Dicionários

- Um dicionário é uma coleção não ordenada capaz de armazenar pares chave-valor
- Um dicionário é criado utilizando { }
- Cada entrada, isto é, cada par chave valor é dado por [chave] : [valor]
- Cada item do dicionário é separado por vírgula (",")
- Para retornar a uma valor associado à uma chave, deve-se utilizar notação `dicionario[chave]`
- As chaves de um dicionário devem ser imutável, além de não poder haver chaves duplicadas

Dicionários

```
In [1]: #Criando um dicionário com pares chave-valor pré-definidos  
dict = {'000.000.000-00' : 'Rafael',  
        '111.111.111-11' : 'Ricardo',  
        '222.222.222-22' : 'Vitor'}
```

```
In [2]: dict
```

```
Out[2]: {'000.000.000-00': 'Rafael',  
        '111.111.111-11': 'Ricardo',  
        '222.222.222-22': 'Vitor'}
```

```
In [3]: dict['000.000.000-00'] #Retornando um valor associado à uma chave
```

```
Out[3]: 'Rafael'
```

Dicionários

- Tentar retornar o valor de uma chave não existente causa uma exceção
- Pode-se verificar de um item existe por meio dos operadores `in` e `not in`
- Pode-se retornar o conjunto de chaves e valores de um dicionário por meio dos métodos `keys()` e `values()`
- **OBSERVAÇÃO:** `in` e `not in` também pode ser utilizado para verificar se elementos pertencem às listas ou conjuntos

Dicionários

```
In [4]: dict['aaaaaa']
```

```
-----  
-----  
KeyError                                Traceback (most recent call  
last)  
<ipython-input-4-8046cc8379d0> in <module>  
----> 1 dict['aaaaaa']  
  
KeyError: 'aaaaaa'
```

```
In [5]: 'aaaaaa' in dict
```

```
Out[5]: False
```

```
In [6]: '000.000.000-00' in dict
```

```
Out[6]: True
```

Dicionários

- Para adicionar um elemento no dicionário
 - Utilizar o mesmo mecanismo de retorno de um valor, porém, realizando uma atribuição
 - Utilizando o método `update()`, cujo conteúdo é uma entrada do dicionário
- Para atualizar uma entrada, pode-se utilizar os mesmos procedimentos da adição
- Para remoção deve-se utilizar a palavra reservada `del` e em seguida a entrada que deseja-se remover

Dicionários

```
In [12]: dict
```

```
Out[12]: {'000.000.000-00': 'Rafael',  
         '111.111.111-11': 'Ricardo',  
         '222.222.222-22': 'Vitor'}
```

```
In [13]: dict['333.333.333-33'] = 'Ronaldo'
```

```
In [14]: dict
```

```
Out[14]: {'000.000.000-00': 'Rafael',  
         '111.111.111-11': 'Ricardo',  
         '222.222.222-22': 'Vitor',  
         '333.333.333-33': 'Ronaldo'}
```

```
In [15]: dict.update({'444.444.444-44': 'Juliano'})
```

```
In [16]: dict
```

```
Out[16]: {'000.000.000-00': 'Rafael',  
         '111.111.111-11': 'Ricardo',  
         '222.222.222-22': 'Vitor',  
         '333.333.333-33': 'Ronaldo',  
         '444.444.444-44': 'Juliano'}
```

```
In [17]: del dict['333.333.333-33']
```

```
In [18]: dict
```

```
Out[18]: {'000.000.000-00': 'Rafael',  
         '111.111.111-11': 'Ricardo',  
         '222.222.222-22': 'Vitor',  
         '444.444.444-44': 'Juliano'}
```

Conjuntos

- Conjuntos representam coleções de elementos de forma que não haja elementos repetidos
- Conjuntos armazenam apenas elementos imutáveis (`strings`, `int`, `floats` ou `tuplas`)
- Para criar um conjunto usa-se `{ }` e os elementos dos conjuntos são separados por vírgula `“,”`
-
- Existem métodos na classe `set` já implementados para operações tradicionais de conjuntos, como união, intersecção e diferença
- Também existem verificações tradicionais, como verificar se um conjunto é subconjunto de outro

Conjuntos

```
In [2]: #Criando um conjunto  
set1 = {1,2,3,3,3,3,3,5,6}
```

```
In [3]: set1 #Perceba que há a eliminação automática de elementos repetidos
```

```
Out[3]: {1, 2, 3, 5, 6}
```

```
In [6]: lista = [1,2,3,3,3,3,3,5,6,8,9,10]
```

```
In [7]: set2 = set(lista) #Criando um conjunto a partir de uma lista
```

```
In [8]: set2
```

```
Out[8]: {1, 2, 3, 5, 6, 8, 9, 10}
```

```
In [9]: set2.add(11) #Adicionando um elemento no conjunto
```

```
In [10]: set2
```

```
Out[10]: {1, 2, 3, 5, 6, 8, 9, 10, 11}
```


Conjuntos

```
In [7]: set1 = {1,2,3,4,5}
        set2 = {4,5,6,7,8}
        set3 = {4,5,6}

In [8]: set1.union(set2) #União do set1 e set2
Out[8]: {1, 2, 3, 4, 5, 6, 7, 8}

In [9]: set1.intersection(set2) #intersecção entre o set1 e o set2
Out[9]: {4, 5}

In [10]: set1.difference(set2) #Retornando a diferença entre o set1 e o set2
Out[10]: {1, 2, 3}

In [11]: set3.issubset(set2)
Out[11]: True

In [12]: set1 |= {14,15,16} #Operação de união de conjuntos

In [13]: set1
Out[13]: {1, 2, 3, 4, 5, 14, 15, 16}

In [14]: set1.add(17) #Adicionando um elemento no conjunto

In [15]: set1
Out[15]: {1, 2, 3, 4, 5, 14, 15, 16, 17}

In [16]: set1.remove(3) #Removendo um elemento no conjunto

In [18]: 1 in set1
Out[18]: True
```

None

- Valores do tipo None são utilizadas para criar variáveis mas não atribuir nenhum valor e consequentemente nenhum tipo à elas
- Outro ponto importante é que testes condicionais em uma variável None sempre resultarão em False

```
In [41]: 1 if resultado == 0:
          2     print('Resultado Inválido')

-----
NameError                                Traceback
(most recent call last)
<ipython-input-41-0e1096d35127> in <module>
----> 1 if resultado == 0:
      2     print('Resultado Inválido')

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

```
In [42]: 1 resultado = None
          2 if resultado == 0:
          3     print('É Verdade!')
          4 else:
          5     print('É Falso!')

É Falso!
```

```
In [43]: 1 resultado = None
          2 if resultado == False:
          3     print('É Verdade!')
          4 else:
          5     print('É Falso!')

É Falso!
```

Operações Matemáticas

- Operadores matemáticos no Python:

- $+$: soma
- $-$: subtração
- $/$: divisão
- $*$: multiplicação
- $\%$: resto da divisão
- $//$: parte inteira da divisão
- $**$: elevação

Operações Matemáticas

```
In [1]: 5 + 10
```

```
Out[1]: 15
```

```
In [2]: 5 * 10
```

```
Out[2]: 50
```

```
In [3]: 5 / 10
```

```
Out[3]: 0.5
```

```
In [4]: 15 % 10
```

```
Out[4]: 5
```

```
In [5]: 15 // 10
```

```
Out[5]: 1
```

```
In [6]: 2 ** 3
```

```
Out[6]: 8
```

Operações Matemáticas

- Quando há operações matemáticas entre tipos numéricos diferentes, o Python realiza a coerção de tipo automática e o resultado será do tipo de maior capacidade de armazenamento.

```
In [3]: 1 num1 = 10
In [4]: 1 type(num1)
Out[4]: int

In [5]: 1 num2 = 10.4
In [6]: 1 type(num2)
Out[6]: float

In [7]: 1 result1 = num1 + num2
In [9]: 1 type(result1)
Out[9]: float

In [10]: 1 num3 = 10.12312443242342342j
In [11]: 1 type(num3)
Out[11]: complex

In [12]: 1 result2 = num2 * num3
In [13]: 1 type(result2)
Out[13]: complex
```

Operações Matemáticas

- O Python aceita a operação de soma entre um tipo numérico e uma *string* como em outras linguagens
- Porém, o Python provê algumas facilidades de sintaxe (*syntax sugar*), como a multiplicação de um inteiro e uma *string*

```
In [15]: 1 str + num
```

```

-----
TypeError                                Traceback (most recent call
  last)
<ipython-input-15-893945584732> in <module>
----> 1 str + num

TypeError: can only concatenate str (not "int") to str

```

Operações Matemáticas

- **OBSERVAÇÃO 1:** Python suporta atribuição composta, i.e., $+=$, $-=$, $/=$, $*=$, $\%=$, $**=$, e $//=$
- **OBSERVAÇÃO 2:** Python **não** possui os operadores unários $++$ e $--$ para incremento e decremento de valores inteiros

E/S Básica

- Entrada básica: função `input([string a ser apresentada ao usuário])`

```
In [1]: idade = input('Digite a sua idade: ')
```

```
Digite a sua idade: 45
```

```
In [4]: idade
```

```
Out[4]: '45'
```


E/S Básica

- A função `input` sempre retorna um *string*
- Portanto é necessária a conversão para um tipo apropriado caso necessário

```
In [1]: idade = input('Digite a sua idade: ')
```

```
Digite a sua idade: 45
```

```
In [4]: idade
```

```
Out[4]: '45'
```

```
In [5]: idade_daqui_10_anos = int(idade) + 10
```

```
In [6]: idade_daqui_10_anos
```

```
Out[6]: 55
```

```
In [7]: type(idade_daqui_10_anos)
```

```
Out[7]: int
```

E/S Básico

- **OBSERVAÇÃO:** quando tiver em dúvida sobre os parâmetros e funcionamento de uma função, basta digitar o nome da função seguindo do símbolo de interrogação ("?")

```
In [1]: input?
```

```
In [ ]:
```

```
Signature: input(prompt='')  
Docstring:  
Forward raw_input to frontends  
  
Raises  
-----  
StdinNotImplementedError if active frontend doesn't support stdin.  
File:      ~/anaconda3/lib/python3.7/site-packages/ipykernel/kernelbase.py  
Type:      method
```

E/S Básico

- Saída básica: função `print(...)`
- Imprimindo só uma *string*

```
In [1]: print('Uba Uba Uba Hey!')
```

```
Uba Uba Uba Hey!
```

E/S Básico

- Associando *strings* e variáveis
 - Estilo printf do C:

```
In [8]: idade = 25  
        nome = 'Rafael'
```

```
In [9]: print('A idade do %s é %d anos' % (nome, idade))  
A idade do Rafael é 25 anos
```

- Separando o conteúdo por vírgulas:

```
In [8]: idade = 25  
        nome = 'Rafael'
```

```
In [10]: print('A idade do', nome, 'é', idade, 'anos')  
A idade do Rafael é 25 anos
```

E/S Básico

- Associando *strings* e variáveis
 - Por padrão, cada argumento da função `print` será separado por um espaço e ao final será impressa uma quebra de linha
 - Pode-se manipular essa configuração por meio dos parâmetros `sep` e `end`

```
In [8]: idade = 25  
        nome = 'Rafael'
```

```
In [11]: print('A idade do', nome, 'é', idade, 'anos', sep='_', end='(FIM DA LINHA)')  
A idade do_Rafael_é_25_anos(FIM DA LINHA)
```

E/S Básico

- Associando *strings* e variáveis
 - Utilizando uma *string* formatada `f'[string content]'`

```
In [8]: idade = 25  
        nome = 'Rafael'
```

```
In [12]: print(f'A idade do {nome} é {idade} anos')
```

A idade do Rafael é 25 anos

```
In [15]: idade = 25  
        nome = 'Rafael'  
        altura = 1.93567
```

```
In [21]: print(f'A idade do {nome} é {idade} anos, e a altura dele é {altura:.2f}')
```

A idade do Rafael é 25 anos, e a altura dele é 1.94

E/S Básico

```
In [4]: tabuada = 2  
        contador = 1  
        while contador <= 10:  
            print(f'{tabuada} x {contador} = {tabuada * contador}')  
            contador += 1
```

```
2 x 1 = 2  
2 x 2 = 4  
2 x 3 = 6  
2 x 4 = 8  
2 x 5 = 10  
2 x 6 = 12  
2 x 7 = 14  
2 x 8 = 16  
2 x 9 = 18  
2 x 10 = 20
```

E/S Básico

```
In [8]: tabuada = 2  
        contador = 1  
        while contador <= 10:  
            print(f'{tabuada} x {contador:2} = {tabuada * contador}')  
            contador += 1
```

```
2 x 1 = 2  
2 x 2 = 4  
2 x 3 = 6  
2 x 4 = 8  
2 x 5 = 10  
2 x 6 = 12  
2 x 7 = 14  
2 x 8 = 16  
2 x 9 = 18  
2 x 10 = 20
```


E/S Básico

```
In [9]: tabuada = 2  
        contador = 1  
        while contador <= 10:  
            print(f'{tabuada} x {contador:<2} = {tabuada * contador}')  
            contador += 1
```

```
2 x 1  = 2  
2 x 2  = 4  
2 x 3  = 6  
2 x 4  = 8  
2 x 5  = 10  
2 x 6  = 12  
2 x 7  = 14  
2 x 8  = 16  
2 x 9  = 18  
2 x 10 = 20
```

E/S Básico

```
In [13]: for num in range(1,11):  
         print(f'{tabuada} x {num:^6} = {tabuada * num}')
```

```
5 x 1    = 5  
5 x 2    = 10  
5 x 3    = 15  
5 x 4    = 20  
5 x 5    = 25  
5 x 6    = 30  
5 x 7    = 35  
5 x 8    = 40  
5 x 9    = 45  
5 x 10   = 50
```

E/S Básico

● OBSERVAÇÕES:

- A f-string só funciona a partir do Python 3.6
- Pode-se utilizar o conceito de string formatada, semelhante à f-string, nas versões anteriores
- O que diferente é que as variáveis que irão compor a string aparecerão dentro de uma função format

```
In [2]: 1 nome = 'Rafael'  
        2 idade = 'idade'
```

```
In [3]: 1 'O nome do fulano é {}, e a idade dele é de {}'.format(nome,idade)
```

```
Out[3]: 'O nome do fulano é Rafael, e a idade dele é de idade'
```

E/S Básico

- **OBSERVAÇÃO:** assim como têm-se a string formatada, têm-se a *raw string*
 - Uma *raw string* é dada por `r'[conteúdo]'`
 - Todo o conteúdo dentro de uma *raw string* é impresso literalmente

```
1 print(r'Vou colocar vários escapes aqui \n \t \r \' ')\n2 print(r'e vocês verão que todos serão \n \n \n impressos')
```

Vou colocar vários escapes aqui \n \t \r '\n e vocês verão que todos serão \n \n \n impressos

Controle de Fluxo

- Estruturas de controles de fluxo são utilizadas para alterar o fluxo sequencial de execução de instruções de um programa
- Para isso podemos fazer uso de:
 - Estruturas de Decisão
 - Estruturas de Repetição
- Ambas fazem uso de operadores lógicos para analisar uma condição e decidir qual trecho do código deve ser executado, ou se um determinado trecho de código deve ser repetido

Testes Lógicos

- Os testes lógicos são compostos por operadores de comparação e podem ser compostos por operadores booleanos
- Os operadores de comparação são: `==`, `!=`, `<`, `>`, `<=`, e `>=`
- Os operadores booleanos são: `and`, `or`, `not`, e `^`

Operadores de Comparação

- Os operadores de comparação podem ter diferentes efeitos de acordo com o tipo de dados de são aplicados
- Nos tipos numéricos, o resultado é simples:

```
In [1]: 1 < 10
```

```
Out[1]: True
```

```
In [2]: 5 > 20
```

```
Out[2]: False
```

```
In [3]: 10 >= 10
```

```
Out[3]: True
```

```
In [4]: 11 != 10
```

```
Out[4]: True
```

```
In [5]: 10 == 10
```

```
Out[5]: True
```

Operadores de Comparação

- OBSERVAÇÃO: o Python provê uma característica diferente de outras linguagens para comparação de intervalo de valores

```
In [1]: nota = 5
```

```
In [2]: 3 <= nota <= 7
```

```
Out[2]: True
```

```
In [3]: nota = 8
```

```
In [4]: 3 <= nota <= 7
```

```
Out[4]: False
```


Operadores de Comparação

- Já nas *strings*, os resultados são interessantes
- Por exemplo, o operador `<=` pode ser utilizado para verificar se a *string* a esquerda é uma *substring* da *string* a direita

```
In [1]: nome1 = 'Rafael Rossi'
```

```
In [2]: nome2 = 'Rafael'
```

```
In [3]: nome3 = 'Rafael Rossi'
```

```
In [4]: nome1 == nome3
```

```
Out[4]: True
```

```
In [5]: nome1 == nome2
```

```
Out[5]: False
```

```
In [6]: nome2 <= nome1
```

```
Out[6]: True
```

```
In [7]: nome1 >= nome2
```

```
Out[7]: True
```

Operadores de Comparação

- Nas listas ou conjuntos, o feito dos operadores de comparação são os mesmos dos obtidos com *strings*

```
In [1]: lista1 = [1,2,3,4]
```

```
In [2]: lista2 = [1,2,3,4,5]
```

```
In [3]: lista3 = [1,2,3,4,5]
```

```
In [4]: lista1 == lista2
```

```
Out[4]: False
```

```
In [5]: lista2 == lista3
```

```
Out[5]: True
```

```
In [6]: lista1 != lista2
```

```
Out[6]: True
```

```
In [7]: lista1 <= lista2
```

```
Out[7]: True
```

```
In [8]: lista2 >= lista1
```

```
Out[8]: True
```

Operadores de Comparação

- Operadores `in` e `not in` podem ser aplicados às listas ou conjuntos para verificar se um elemento está contido nestas

```
In [1]: numeros = [1,3,5,7,9,11]
```

```
In [2]: 3 in numeros
```

```
Out[2]: True
```

```
In [3]: 4 in numeros
```

```
Out[3]: False
```

```
In [5]: 4 not in numeros
```

```
Out[5]: True
```

Operadores Booleanos

- Operadores booleanos em Python
 - **and**: E lógico
 - **or**: Ou lógico
 - **not**: Negação lógica
 - **^**: Ou exclusivo lógico

Operadores Booleanos

```
In [1]: nome = 'Rafael'  
        profissao = 'Professor'  
        idade = 34
```

```
In [4]: idade > 30 or profissao == 'Professor'
```

Out[4]: True

```
In [5]: idade > 30 or profissao == 'Go-go boy'
```

Out[5]: True

```
In [6]: idade > 30 and profissao == 'Go-go boy'
```

Out[6]: False

```
In [7]: not idade > 30 or profissao == 'Go-go boy'
```

Out[7]: False

Operadores Booleanos

```
In [1]: True == 1
```

```
Out[1]: True
```

```
In [2]: True == 0
```

```
Out[2]: False
```

```
In [3]: False == 1
```

```
Out[3]: False
```

```
In [4]: False == 0
```

```
Out[4]: True
```

```
In [5]: True == 10
```

```
Out[5]: False
```

Operadores Booleanos

```
In [1]: nome = 'Rafael'  
        profissao = 'Professor'  
        idade = 34
```

```
In [3]: (nome == 'Rafael') ^ (profissao == 'Professor')
```

```
Out[3]: False
```

```
In [4]: (nome == 'Rafael') ^ (profissao == 'Patrão')
```

```
Out[4]: True
```

Estruturas de Seleção

- Python não possui a estrutura `switch-case`, comum em outras linguagens
- Como estruturas de decisão têm-se:
 - If e suas variantes
 - Expressão condicional

Estruturas de Seleção

- If simples → `if [condicao] : [ação]`

```
In [1]: nota = 5
```

```
In [7]: """Essa estrutura de if pode ser utilizada caso haja uma única
instrução a ser executada"""
if nota <= 6: print('Reprovado')
```

Reprovado

Estruturas de Seleção

- If com bloco de instruções
- Um bloco de instruções é dado por todos os comandos que estiverem identados após o símbolo “:”

```
In [1]: nota = 5
```

```
In [8]: if nota <= 6:  
        print('Reprovado')  
        print('Teste novamente o próximo semestre')
```

```
Reprovado  
Teste novamente o próximo semestre
```

- **OBSERVAÇÃO:** a indentação no Python é dada por quatro espaços em branco (o pressionamento da tecla TAB nos editores causa o mesmo efeito)

Estruturas de Seleção

- Erros na indentação podem causar comportamentos indesejados

```
In [12]: nota = 8
```

```
In [13]: if nota < 6:  
         print('Reprovado')  
         print('Teste novamente o próximo semestre')
```

Teste novamente o próximo semestre

Estruturas de Seleção

- Erros na indentação podem causar exceções

```
In [12]: nota = 8
```

```
In [14]: if nota < 6:  
        print('Reprovado')  
        print('Teste novamente o próximo semestre')
```

```
File "<ipython-input-14-e8430069ccda>", line 3  
    print('Teste novamente o próximo semestre')  
    ^
```

```
IndentationError: unexpected indent
```

Estruturas de Seleção

- If-else

```
In [1]: nota = 5
```

```
In [10]: if nota < 6:  
         print('Reprovado')  
         else:  
         print('Aprovado')
```

Reprovado

Estruturas de Seleção

- If-elif-else

```
In [12]: nota = 8
```

```
In [16]: if nota < 3:
          print('Reprovado')
          elif nota < 6:
          print('Em recuperação')
          else:
          print('Aprovado')
```

Aprovado

Estruturas de Seleção

- Expressão condicional: função que retorna um valor mediante uma análise de condição

```
In [12]: nota = 8
```

```
In [18]: resultado = ('passou' if nota >= 6 else 'reprovou')
```

```
In [19]: print('O aluno', resultado)
```

O aluno passou

- Expressão condicional encadeada

```
In [22]: nota = 5
```

```
In [23]: resultado = ('passou' if nota < 3 else ('pode recuperar' if nota < 6 else 'passou'))
```

```
In [24]: print('O aluno', resultado)
```

O aluno pode recuperar

Estruturas de Repetição

- As estruturas de repetição do Python diferem do que é comum em algumas languages
- Não existe `do-while`
- Já o `for` não considera uma variável de controle, e consequentemente não faz testes condicionais e incrementos na mesma \Rightarrow apenas é utilizado para percorrer uma lista de elementos iteráveis

Estruturas de Repetição

- While

- Estrutura de repetição com teste condicional no início
- Notação: `while [condição] : ...`

```
In [26]: tabuada = 3  
        contador = 1
```

```
In [27]: while contador <= 10:  
        resultado = tabuada * contador  
        print(f'{tabuada} x {contador:<2} = {resultado}')
```

```
        contador += 1  
  
3 x 1  = 3  
3 x 2  = 6  
3 x 3  = 9  
3 x 4  = 12  
3 x 5  = 15  
3 x 6  = 18  
3 x 7  = 21  
3 x 8  = 24  
3 x 9  = 27  
3 x 10 = 30
```

Estruturas de Repetição

- For
 - Utilizado para percorrer elementos iteráveis
 - A variável do for a cada iteração irá assumir um valor do conjunto iterável
 - Notação: `for [variável] in [iterável]: ...`

```
In [28]: cidades = ['Porto Ferreira', 'Três Lagoas', 'Campo Grande', 'Andradina']
```

```
In [29]: for cidade in cidades:  
         print('Nome da cidade:', cidade)
```

```
Nome da cidade: Porto Ferreira  
Nome da cidade: Três Lagoas  
Nome da cidade: Campo Grande  
Nome da cidade: Andradina
```

Estruturas de Repetição

- Função `range()`: possibilita utilizar o `for` de maneira semelhante ao tradicional (com um variável que será incrementada a cada iteração)
- Função `range(ini,fim)`: gera um conjunto de valores a parte de *ini*, com incremento unitário, até *fim* - 1

```
In [4]: range1 = range(1,10) #Criando um range sequencial (não inclui o último elemento)
```

```
In [5]: range2 = range(1,10,2) #Criando um range com passo (definido pelo 3º argumento)
```

```
In [6]: for valor in range1:  
        print(valor, end=' ')
```

1 2 3 4 5 6 7 8 9

```
In [7]: for valor in range2:  
        print(valor, end=' ')
```

1 3 5 7 9

Estruturas de Repetição

- Combinando o `for` com a função `range` para imprimir os índices e os elementos de uma lista

```
In [28]: cidades = ['Porto Ferreira', 'Três Lagoas', 'Campo Grande', 'Andradina']
```

```
In [30]: for pos in range(0, len(cidades)):
         print(pos, '-', cidades[pos])
```

```
0 - Porto Ferreira
1 - Três Lagoas
2 - Campo Grande
3 - Andradina
```

Estruturas de Repetição

- Combinando o `for` com a função `enumerate` para imprimir os índices e os elementos de uma lista

```
In [28]: cidades = ['Porto Ferreira', 'Três Lagoas', 'Campo Grande', 'Andradina']
```

```
In [33]: enum = enumerate(cidades)
```

```
In [35]: list(enum)
```

```
Out[35]: [(0, 'Porto Ferreira'),  
          (1, 'Três Lagoas'),  
          (2, 'Campo Grande'),  
          (3, 'Andradina')]
```

```
In [44]: for num, ele in enumerate(cidades):  
          print(num, '-', ele)
```

```
0 - Porto Ferreira  
1 - Três Lagoas  
2 - Campo Grande  
3 - Andradina
```

Estruturas de Repetição

- For em um dicionário \Rightarrow pode-se utilizar o método `dict.items()` que retorna uma lista de tuplas
- O primeiro elemento de cada tupla é uma chave e o segundo é um valor

```
In [10]: dict.items()
```

```
Out[10]: dict_items([('000.000.000-00', 'Rafael'), ('111.111.111-11', 'Ricardo'), ('222.222.222-22', 'Vitor')])
```

```
In [11]: for key, value in dict.items():  
         print(key, '-', value)
```

```
000.000.000-00 - Rafael  
111.111.111-11 - Ricardo  
222.222.222-22 - Vitor
```

Estruturas de Repetição

- O Python também provê as estruturas `break` e `continue` para manipular estruturas de repetição

```
In [4]: 1 for num in range(1,10):  
        2     if(num == 5):  
        3         continue  
        4     print(num)
```

```
1  
2  
3  
4  
6  
7  
8  
9
```

Estruturas de Repetição

In [5]:

```
1 num_chute = 5
2 while(True):
3     num_usuario = int(input('Digite um número:'))
4     if(num_usuario == num_chute):
5         print('Acertou abestado!')
6         break
7     else:
8         print('Número errado')
```

```
Digite um número:4
Número errado
Digite um número:6
Número errado
Digite um número:5
Acertou abestado!
```


List Comprehensions

- Há também a possibilidade de inicializar listas utilizando o conceito de *List Comprehensions*
- As *List Comprehensions* permitem especificar um loop para gerar valores
- Permite também especificar testes condicionais para a geração dos valores

List Comprehensions

```
In [34]: lista = []
```

```
In [35]: lista
```

```
Out[35]: []
```

```
In [41]: lista = [num for num in range(1,10)]
```

```
In [42]: lista
```

```
Out[42]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [43]: lista = [num*2 for num in range(1,10)]
```

```
In [44]: lista
```

```
Out[44]: [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
In [47]: lista = [num*2 for num in range(1,10) if num*2 % 4 == 0]
```

```
In [48]: lista
```

```
Out[48]: [4, 8, 12, 16]
```

Material Complementar

- Wikipedia - Python

<https://pt.wikipedia.org/wiki/Python>

- Curso em Vídeo - Python

<https://www.youtube.com/channel/UCrWvhVmt0Qac3HgsjQK62FQ>

- Python Tutorial

<https://www.devmedia.com.br/python-tutorial/33274>

Conteúdo Complementar

- Capítulo 2 - O que é Python <https://www.caelum.com.br/apostila-python-orientacao-objetos/o-que-e-python/>
- Capítulo 3 - Variáveis e tipos embutidos
<https://www.caelum.com.br/apostila-python-orientacao-objetos/declarando-e-usando-variaveis/>
- Capítulo 4 - Estrutura de dados
<https://www.caelum.com.br/apostila-python-orientacao-objetos/estrutura-de-dados/>

Material Complementar

- Curso de Python 3 - Mundo 1: Fundamentos

https://www.youtube.com/playlist?list=PLHz_AreHm4d1KP6QQCekuIPky1CiwmdI6

- Curso de Python 3 - Mundo 2: Estruturas de Controle

https://www.youtube.com/playlist?list=PLHz_AreHm4dk_nZHmxxf_JOWRAqy5Czye

- Curso de Python 3 - Mundo 3: Estruturas Compostas

https://www.youtube.com/watch?v=0LB3FSfjvao&list=PLHz_AreHm4dksnH2jVTIVNviIMBVYyFnH

- Exercícios de Python 3

https://www.youtube.com/watch?v=nIHq1MtJaKs&list=PLHz_AreHm4dm6wYOIW20Nyg12TAjmMGT-

Imagem do Dia



Tópicos em Inteligência Artificial

<http://ava.ufms.br/>

Rafael Geraldeli Rossi
rafael.g.rossi@ufms.br