UNIVERSIDADE FEDERAL DO PARÁ

INSTITUTO DE TECNOLOGIA - ITEC

Apostila do Curso de introdução ao Python3

BELÉM - PA

2019

Sumário

1. Apresentação

Este material foi desenvolvido para apoiar o curso de introdução a Python3 ministrado na semana do ITEC. O material é um guia introdutório sobre a linguagem e suas aplicações para visualização de dados, manipulação de matrizes e vetores e métodos numéricos.

O material estará disponibilizado no Github, assim como os códigos que compuseram o curso. Dessa forma sendo disponibilizado para o acesso democrático de todos.

Para sanar duvidas relativas ao curso, ou ao material, ou em relação a linguagem Python e alguma aplicação, estará disponível para contato os e-mails dos ministrantes do curso.

Caso haja algum comentário, duvida ou opinião sobre o curso ou o material, mande por e-mail ! Nós agradeceríamos o feedback da comunidade do ITEC.

1.1. Contatos e links

- Contato: Rodrigo Gomes Dutra

  e-mail: dutra.rgdgd@gmail.com

  github: https://github.com/rodgdutra

- Contato: Antonio Adrian

e-mail: antonioadrian.alves@gmail.com

github: https://github.com/AntonioAdrian

- Link do material do curso e dos códigos:

  https://github.com/rodgdutra/minicurso_python

2. **Introdução**

Este material foi desenvolvido para apoiar o curso de introdução a Python3 ministrado na semana do ITEC. O material é um guia introdutório sobre a linguagem e suas aplicações para visualização de dados, manipulação de matrizes e vetores e métodos numéricos.

Python é uma linguagem de propósito geral de altíssimo nível (VHLL – Very high Level Language), criada pelo holandês Guido Van Rossum seguindo o idel de “Programação de Computadores para todos”. Esta linguagem é multiparadigma, suporta o paradigma orientado a objetos, imperativo, funcional e procedural. Possui tipagem dinâmica e uma de suas principais características é permitir a fácil leitura do código e exigir poucas linhas de código se comparado ao mesmo programa em outras linguagens. Devido às suas características, ela é principalmente utilizada para processamento de textos, dados científicos e criação de CGIs para páginas dinâmicas para a web. Foi considerada pelo público a 3ª linguagem "mais amada", de acordo com uma pesquisa conduzida pelo site Stack Overflow em 2018,e está entre as 5 linguagens mais populares, de acordo com uma pesquisa conduzida pela RedMonk.

A linguagem foi projetada com a filosofia de enfatizar a importância do esforço do programador sobre o esforço computacional. Prioriza a legibilidade do código sobre a velocidade ou expressividade. Combina uma sintaxe concisa e clara com os recursos poderosos de sua biblioteca padrão e por módulos e frameworks desenvolvidos por terceiros.

As Aplicações de python vão desde o backend até ciência de dados(data science) e machine learning. A Imagem abaixo, segundo o site https://data-flair.training/blogs/python-applications/, ilustra as principais aplicações do python3 até então:

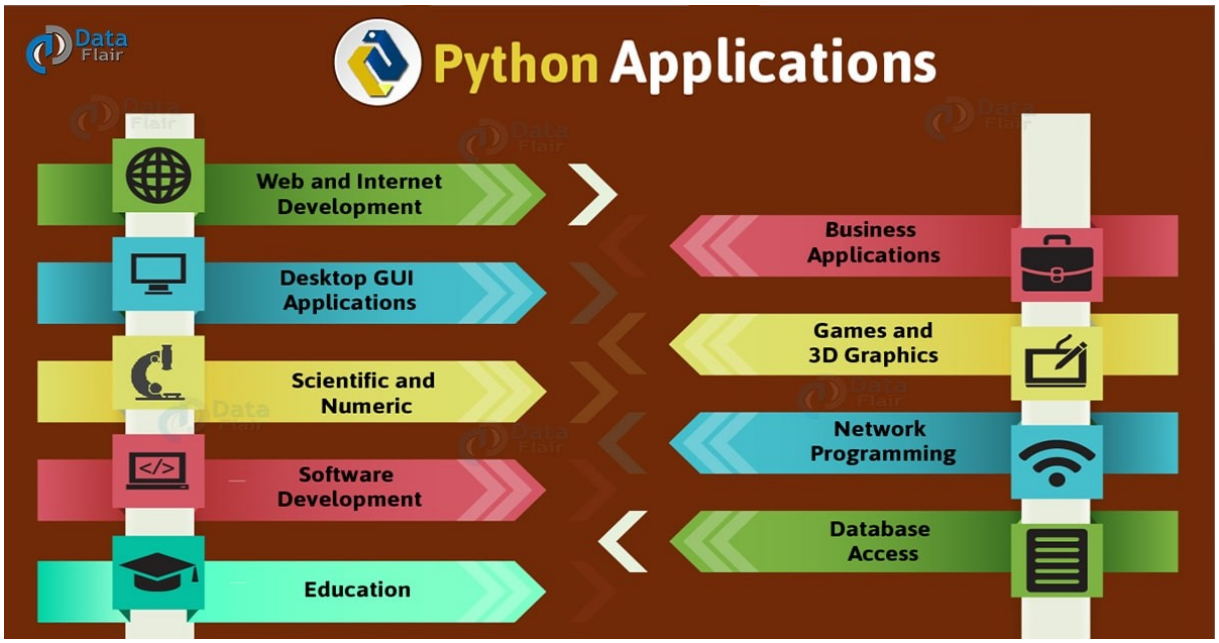


Figura. 1. Real World Applications of Python Programming.

Além dessas aplicações, ainda há aplicações em IOT, como na programação de scripts para raspberry Pi.

Baseado no indice de TIOBE, o python já consta como a terceira linguagem de programação mais utilizada.

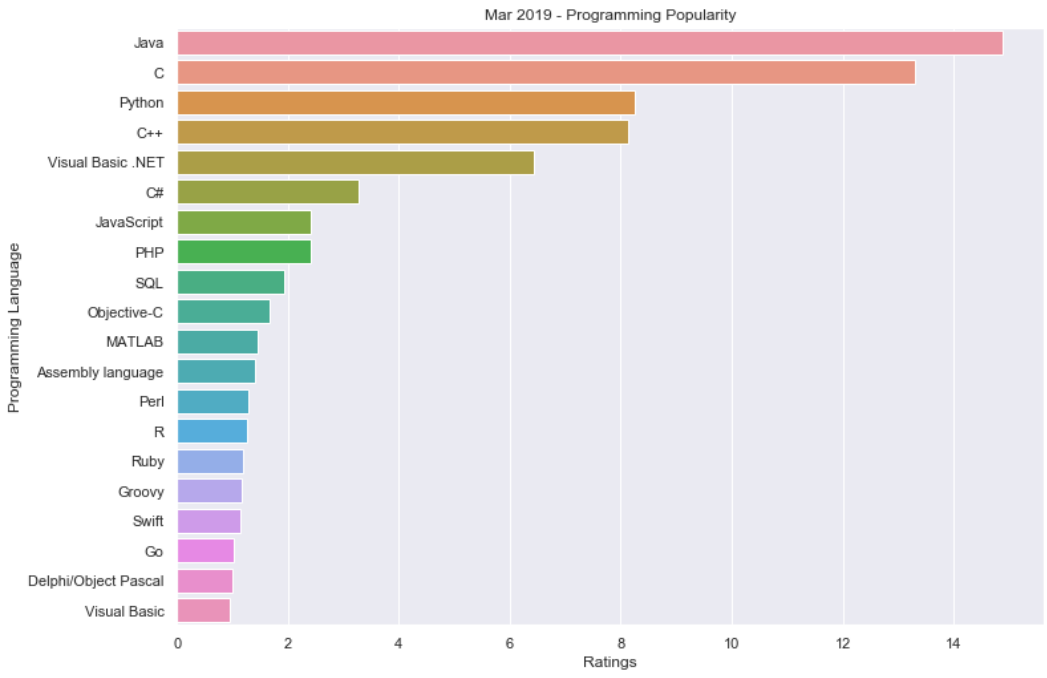


Figura. 2. Linguagens de programação mais utilizadas.

Dessa forma, além do Python ser uma linguagem de programação multiparadigma, é uma linguagem estremamente utilizada no mundo todo com diversas aplicações do “ Mundo real “, assim não faltam motivos para estudar essa linguagem e aplica-la de diversas maneiras.

3. Instalação de um ambiente de desenvolvimento Python

Para instalar o Python3 é recomendado baixar o pacote de instalação direto do link: https://www.python.org/downloads/

Após a instalação do Python3 é recomendado um conjunto de ferramentas para o desenvolvimento com a linguagem. A primeira dessas ferramentas é o Spyder, o qual possui uma interface que lembra a do matlab e possibilita a utilização do console python juntamente com a possibilidade de rodar e escrever scripts, realizar o processo de debug e visualizar variáveis.

Outra ferramenta interessante é o Jupyter notebook, que possibilita ao usuário escrever o codigo, roda-lo e visualizar gráficos na mesma página, como um caderno de anotações. Além disso os notebooks com os gráficos já gerados podem ser salvos, dessa forma para visualização posterior o gráfico já estará plotado na pagina do notebook.

Ambas as ferramentas são instaladas no windows pelo pacote Anaconda, no link: https://www.anaconda.com/distribution/.

Com o Anaconda instalado o usuário tem acesso as ferramentas listadas e conjuntos de bibliotecas que serão abordados nesse curso para manipulação de dados e visualização de dados, as quais são o **numpy** e a **matplotlib.**

4. Primeiros passos com o Python

Após a instalação do ambiente de desenvolvimento, é necessário a visualização do terminal python como um primeiro passo do usuário que aspira o desenvolvimento no mesmo. Para isso vá no Iniciar e procure por python3, e abra o terminal do python3. Com isso você já pode realizar pequenas operações com o terminal e testar a sintaxe, tente a soma e multiplicação de valores numéricos, dessa forma o terminal do python pode ser utilizado como uma calculadora.

5. Hello Wolrd

Uma maneira muito eficiente e consagrada de introduzir a sitaxe de uma nova linguagem é escrevendo um programa básico um para printar na tela uma mensagem (Hello Wolrd), esse primeiro código vai ajudar no entendimento da filosofia por trás da sintaxe empregada, no caso do python esse primeiro código é escrito da seguinte maneira :

```
print("Hello World")
```

Notamos que o python dá preferência para uma sintaxe clara, explicita, direta e de fácil entendimento, esses conceitos podem ser encontrados mais explicitamente no "*Zen of Python"*, essa caracteristica ficam mais evidentes ainda quando comparamos outros *Hello World*'s de linguagem com C :

```
<stdio.h>
int main(){
   printf("Hello World");
  return 0;
    }
```

Ou Java:

```
public class hello {
public static void main (String arg []){
    System.out.println("hello world");
    }}
```

6. Variáveis

As variáveis em Python são dinamicamente tipadas, isso significa que não precisamos declarar explicitamente o tipo da variável para usa-lá e que ela pode mudar de tipo dependendo do valor atribuido. O python tem quatro tipos primitivos de dados ***int***, ***float***, ***bool*** e ***str***, alem das **lista**, **tuplas** e **dicionários** que seram abordadas mais adiante. Usando o *Spyder*, na janela *Console IPython*, é possível visualizar essas caracteristicas usando a função *type()* que retorna o tipo da variável, como segue:

```
In [1]: A = 5

In [2]: type(A)
Out[2]: int

In [3]: A = 5.5

In [4]: type(A)
Out[4]: float

In [5]: A = True

In [6]: type(A)
Out[6]: bool

In [7]: A = 'True'

In [8]: type(A)
Out[8]: str
```

6.1. Receber valores de entrada e exibir valores no console

Com a função *input()* podemos receber valores digitados pelo *console*, sempe se atentando para o fato da entrada receber o tipo *str*:

```
Var = input("Digite um valor de entrada:")
print(Var)
print(type(Var))
```

Como comumente é desejável receber valores de outros tipo, não apenas *str,* para isso é possível realizar uma conversão de tipo (antes de realizar a conversão, consultar a documentação):

```
Var = input("Digite um valor:")
Var_int=int(Var)
Var_float=float(Var)
Var_bool=bool(Var)


print(Var_int)
print(Var_float)
print(Var_bool)
```

Existem 3 formas de exibir o valor de um variável em Python, passando a variável para a função print ,usando o método *format* ou pelo operador % e o tipo da variável:

```
# Sendo A uma varável do tipo int
print(A)
print("Valor de A :{}".format(A)))
print("Valor de A :%i" %(A))
```

Note que quanto é desejado colocar o valor da variável ao longo de uma frase, se torna mais conveniente usar o método *format,* porem quando o objetivo for apenas visualisar o valor da variável, apenas passar a variável como argumento se torna mais cômodo.

6.2. Operadores matemáticos

O Python suporta todas operações aritimeticas básicas, potenciação, divisão inteira e resto da divisão, cujas sintaxes são:

```
# Sejam A e B duas variáveis quaisquer
A+B     # soma de A e B
A-B     # subtração de A por B
A/B     # divisão de A por B
A*B     # multiplicação de A e B
A**B    # A elevado a B
A//B    # divisão inteira de A por B
A%B     # resto da divisão de A por B
```

Além das operações matemáticas, o Python tambem suporta operações booleanas com as quais é possível escrever qualquer equação booleana:

```
# Sejam A e B duas variáveis booleanas
A and B     # operação and
A or B      # operação or
not A       # operação not
```

6.3. Listas, tuplas e dicionários

Listas são similares a arrays. Como já foi dito, Python é um limguagem dinamicamente tipada, não por acaso os elementos das lista não precisão ser do mesmo tipo como em outras linguagem:

```
lista1 = []                             #criando uma lista vazia
lista2 = [1,2.2,True,"Hello World"]     #criando uma lista com valores iniciais

lista1.append(1)                        #adicionando valores a lista1
lista1.append(2.2)
lista1.append(True)
lista1.append("Hello World")

print(lista1)
print(lista2)
print(lista1[1])        #printando elemento da lista
print(lista2[1:3])      #printando intervalo da lista
print(lista2 + lista1) #concatenando listas
```

Tuplas são similares as lista, a diferença entre essas duas estruturas é que os elementos das tuplas não pode ser modificados, sendo assim os seus valores devem ser setados logo que a variável for declarada:

```
tupla1 = (1, 2.2, False, "Hello")
print(tupla1[1:3])       #printando intervalo da tupla1
tupla1[3] = "Hello World"   #retornará um erro
```

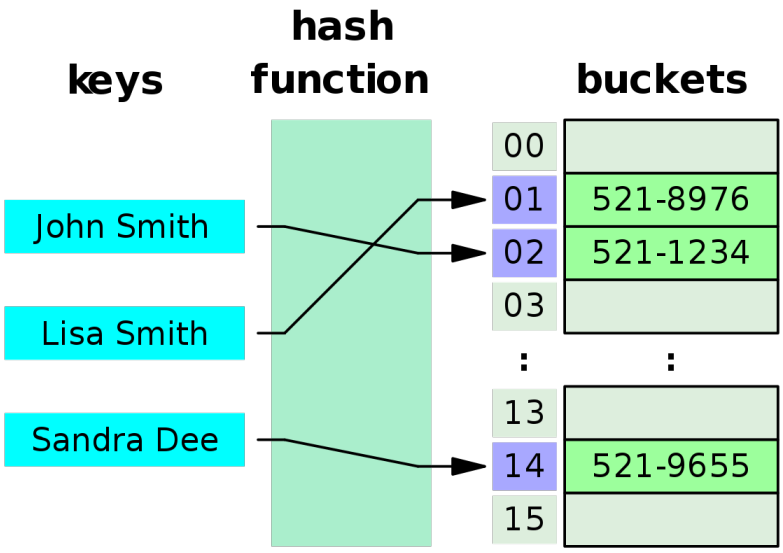


Figura 3: Estrutuda de uma hash table

Dicionários são, em Python, a implementação de uma estrutura de dados conhecida como *hash table*(tabela de dispersão ou espalhamento), essa estrutura é muito importante quando se está trabalhando com grandes bancos de dados por conta do fato de ser possível realizar a busca de uma determinada informação realizando apenas uma operação, sem haver a necessidade de percorrer todo o banco.

O “coração” da *hash table* é a função de hash, pois com ela podemos mapear cada chave em uma posição do array de dados, como mostrado na Figura 3 cada nome(chave) é mapeado através da função de hash em uma determinada posição do vetor que contem o número de celular (valor) associado àquele nome.

A sintaxe utilizada para criar e manipular dicionários em Python é a seguinte:

```
# Criando dicionário e atribuindo chaves e valores
email = {
    "adrian" : "antonioadrian.alves@gmail.com",
    "rodrigo" : "dutra.rgdgd@gmail.com"
}

# Duas maneiras de deletar um valor de um
email.pop('adrian')
del email['rodrigo']

git = {} # criando dicionário vazio

#Armazenando valores no dicionário
git["adrian"] =  "github.com/AntonioAdrian"
git['rodrigo'] = 'github.com/rodgdutra'
```

7. **Primeiros passos com o Numpy**

O módulo numpy é uma das bibliotecas preferidas para trabalhar com matrizes, vetores contendo várias funções para análises numéricas de forma computacionalmente otimizada.

7.1. Alocando vetores e realizando operações

```
# Importando o numpy
import numpy as np

# Alocando um vetor de 0 até 45 com intervalos de 5 em 5
x1 = np.arange(0,50,5)
print("x1            : {0}".format(x1))

# Criando um array com valores randômicos
x2 = np.random.rand(10)
print("x2            : {0}".format(x2))

#Criando um array só com valores 0
x0 = np.zeros(10)
print("x3            : {0}".format(x0))

# Note que o ultimo valor é o final - intervalo
```

As operações com vetores utilizando-se o numpy é realizada de forma facilitada, o fragmento de código abaixo exemplifica as operações matemáticas básicas aplicadas termo a termo de um vetor.

```
# Somando 2 a todos os campos do vetor
soma = x1 + 2
print("soma          : {0}".format(soma))

# Multiplicando cada campo por 0.5
mult = x1 * 0.5
print("multiplicação : {0}".format(mult))

# Somando um array com outro array
x2   = np.arange(0,10)
soma = x1 + x2
print("x1 + x2       : {0}".format(soma))
```

```
# Divindo um array por outro
mult = x2 * x1
print("x2 * x1       : {0}".format(mult))
```

O acesso a posições de vetores é semelhante a sintaxe do C e derivados, como exemplificado no código abaixo:

```
# Acessando valores individuais de vetores
print("x1[0] : {0}".format(x1[0]))

# Nota-se que pode-se alocar valores para os valores acessados
x1[0] = 10
print("x1[0] : {0}".format(x1[0]))
```

7.2. Interações com vetores

O Python permite algumas interações com vetores com uma sintaxe especifica para tal, essa sintaxe permite operações como particionar vetores e selecionar elementos de passo a passo.

```
import numpy as np
x = np.arange(0,50)

# A notação do python para o particionamento de arrays é [inicio:final:passo]
# Nota-se que não necessariamente é necessário utilizar todos os ':'
# partindo o array em 2 metades
x1 = x[:25]
x2 = x[25:50]
print("x1  : {0}".format(x1))
print("x2  : {0}".format(x2))

# Realizando o 'Downsampling' do array
x3 = x[::2]
print("x3  : {0}".format(x3))
```

O modulo numpy também tem funções para operações com vetores, como juntar 2 vetores:

```
# Juntando 2 arrays
x4 = np.concatenate((x1,x2))
print("x4  : {0}".format(x4))
```

Por padrão existem métodos utilizando as variáveis do tipo lista no python para incrementar a lista dinamicamente. Para exemplificar isto e o fato de que é possível passar uma array do tipo *numpy.array* para lista em python o fragmento de código abaixo foi feito.

```
# Transformando um array do tipo numpy para lista do python
x1_list = x1.tolist()
print("classe antes: ",type(x1))
print("classe depois: ",type(x1_list))

# Adicionando termos em uma lista python
x1_list.append(255)
print("x1_list: {0}".format(x1_list))
```

Além das operações matemáticas é possível ainda comparar arrays utilizando-se do operador de comparação de igualdade do python:

```
# Comparando 2 arrays
comp = (x4 == x)
print("comp: {0}".format(comp))
```

Para acessar informações sobre a dimensão do array e seu tamanho é utilizado-se a funçaõ *len*, e se tratando de um array do tipo numpy é possível ver as dimensões através do atributo shape do array.

```
# Acessando informações do array
len_x1 = len(x1)
dim_x1 = x1.shape
```

```
print("tamanho x1: {0}, dimensões x1: {1}".format(len_x1,dim_x1))
```

7.3. Alocando matrizes e primeiras operações

O código disposto abaixo exemplifica a sintaxe do python para a declaração de matrizes, a qual cada linha é separada por **[ ]** e **,** dentro de outro par de **[ ]**. Dessa forma cada linha da matriz é um elemento dentro de um array.

```
import numpy as np
from pprint import pprint
# Nota :  a biblioteca pprint serve para mostrar na tela de
# forma mais agradável matrizes e outras estruturas

# Declarando uma matriz e visualizando-a
A = np.array([[2, 4], [5, -6]])
print("A:")
pprint(A)
```

Para o acesso de dados de uma matriz é importante lembrar que cada linha da matriz é um elemento dentro da estrutura maior. O acesso de uma coluna da matriz o python utiliza de uma sintaxe própria para isso.

```
# Visualizando somente a primeira linha
print("A[0]:")
pprint(A[0])

# Visualizando somente a primeira coluna
print("A j1:")
pprint(A[:,0])
```

Similarmente a vetores no numpy, as matrizes que são vetores multidimensionais são facilmente operadas, graças ao tipo de *numpy.array.*

```
B = np.array([[3,4],[6,9]])
# Soma
print("A + B :")
pprint(A + B)
```

```
# Multiplicação
print("A * B :")
pprint( A * B)

# Transposição
print("A_t:")
pprint(np.transpose(A))
```

As dimensões de matrizes no tipo de *numpy.array* podem ser facilmente modificadas utilizando métodos do tipo *numpy.array:*

```
# Transformando matriz em vetor unidimensional
tran_a = A.flatten()
print("A unidimensional :")
pprint(tran_a)

# Manipulando as dimensões de um array
C = np.arange(0,4)
C = C.reshape(2,2)
print("C :")
pprint(C)
```

7.4. Exercício envolvendo as operações vistas

Exercício: Escreva um código que execute as instruções abaixo, utilizando somente o que foi exposto nos códigos anteriores:

- Aloque um vetor x com 10 elementos, começando do 0
- Aloque um vetor y com 10 valores randômicos
- Aloque uma matriz com 2 colunas e 10 linhas
- Passe os valores de x e y para a matriz, de forma que os valores de x fiquem na primeira coluna e de y fiquem na segunda coluna.
- Mostre na tela a matriz com os valores de x e y

- Adicione um valor escalar para cada valor da coluna y da matriz
- Mostre a matriz novamente

Resolução:

```
import numpy as np
from pprint import pprint
# declarando um vetor de x
x = np.arange(0,10)

# declarando um vetor y de valores randomicos
y = np.random.rand(10)

# Alocando um array de 0
z = np.zeros(20)
z = z.reshape(10,2)
# Colocar a primeira coluna como x
z[:,0] = x
# Colocar a segunda coluna como y
z[:,1] = y
# Mostrando na tela a matriz
print("Matrix xy : ")
pprint(z)
# Adicionando valores para a coluna y
z[:,1] = z[:,1] + 4
# Mostrando na tela a matriz
print("Matrix xy : ")
pprint(z)
```

8. Utilizando funções e declarando-as

O python3 tem uma sintaxe própria para declarar funções e na utilização de funções de decisão como o **if** e **else**, e funções de repetição como o **for** e **while**. Diferentemente do C e do Java, que dependem de estruturas como o { } para definir o que está dentro de uma função em sua declaração ou na sua utilização(**if,else,for,while**), o python3 tem uma sintaxe baseada na sua **identação**, ou seja, na quantidade de **espaços** ou **tabs** que o usuário escreve no código. O código abaixo exemplifica essa estrutura do python baseado na identação.

8.1. Utilizando o **If** e **Else**

```
# Alocando um valor boleano a x
x = True
# Para marcar o final da declaração do if é utilizado o :
# Para demarcar o que estará dentro do if, o seu corpo, é utilizado um tab
# Nota-se que 4 espaços forma-se um tab.
if x == True:
    print("X é True")

# Utilizando if e else
# Para definir um else depois do if
if x == True:
    print("X é True")
else:
    print("X é False")

# If e Else encadeado
# Para definir operações de if e else juntas é utilizado o elif
x = 5
if x == True:
    print("X é True")
elif x == False:
    print("X é False")
else:
    print("X não é boleano")
```

Além do uso clássico do if, else é possível utilizar essas estruturas de decisão para alocar valores a uma variável, para tal o python tem uma sintaxe espessífica, como é exemplificado abaixo

```
# Utilização do If e else na declaração de variáveis
y = 5 if x==True else 6
print("y é {0}".format(y))
x = True
y = 5 if x==True else 6
print("y é {0}".format(y))

# Nota-se que nesse tipo de utilização é necessário o if e o else,
# caso somente o if irá dar erro
```

8.2. Utilizando **for** e **while**

O **For** no python é utilizado para realizar a iteração entre termos de objetos como listas, tuplas, dicionários ou outros tipos de objetos iteráveis.

```
import numpy as np
# Declarando uma lista vazia
x = list() # também poderia ser x = []

# Utilizando o for para adicionar valores em x
for i in range(0,10):
    x.append(i)
print("x : ",x)
# Nota-se que o for é utilizado para iterar em objetos iteraveis
# o objeto iterável nesse caso é o range que cria uma serie
# iteravel do tipo range.

# Utilizando o for em uma lista de strings
animais = ["macaco","cachorro","gato"]

for i in animais:
    print("animal da vez : ", i)
```

Assim como as estruturas de decisão, o for pode ser utilizado para alocar valores a variáveis, iterando em uma lista e realizando operações desejadas termo a termo e salvando na variável declarada.

```
# O for no python pode ser utilizado também na declaração de variáveis
y = np.arange(0,10)

x = [i*(i-1) for i in y]
print("x : ",x)
# dessa forma o for pode ser utilizado para declarar uma lista, realizando
# operações para cada iteração e salvando o resultado na lista em 1 linha só
```

O *while* no python é utilizado da seguinte forma, enquanto a operação entre os parênteses for verdadeira o while continuará realizando a operação dentro de seu corpo, a não ser que seja utilizado o parametro break. O código abaixo exemplifica o uso dessa estrutura:

```
# Utilizando o while
i = 0
while(i < 10):
    print("valor do i: ",i)
    i = i + 1

# Utilizando o break para parar loops
# Criando um loop intermitente por natureza e parando-o com o break
i = 0
while True:
    print("valor do i: ",i)
    i = i + 1
    if i == 10:
        break
```

8.3. Definindo funções utilizando python

Como falado anteriormente, o python possui uma sintaxe diferente para a a definição do corpo nas funções de decisão e de repetição. Na declaração a sintaxe permanece quase a mesma. A exemplo disso está o código abaixo:

```
import numpy as np
# para definir uma função utiliza-se o def
# Os argumentos da função devem estar entre ()
# O corpo deve estar em um bloco de código identado por tab

# definindo uma função para retornar a soma dos elementos de um array
def somatorio(a):
    soma = 0
    for i in a:
        soma = soma + i
    return soma # O return é necessário para retornar o dado

# Definindo uma função principal
# Nota-se que esse passo não é realmente necessário, porém organizar um
# codigo com uma função principal e outras secundárias é o ideal para
# a organização do codigo
def main():
    x = np.arange(10)
    soma = somatorio(x)
    print("soma :", soma)

# Esse bloco do if __name__ == é necessário para chamar a função main
if __name__ == "__main__":
    main()
```

Nota-se que o código acima utiliza uma função **main()** o que é um passo não necessário. Esse passo é somente necessário para manter uma organização concisa do código, dessa forma é possível reconhecer a rotina principal de um script python rapidamente, mesmo se for um script com muitas linhas.

8.4. Definindo funções em uma linha

No python também é possível declarar uma função em uma linha só, para essa tarefa é utilizado o **lambda.** O código abaixo exemplifica esse tipo de uso da função lambda.

```
# Primeiramente, definindo uma função da forma tradicional
# Função para calcular o quadrado de um número:

def quadrado(x):
    return x*x

# Função main
def main():
    x = 2
    x_2 = quadrado(x)

    # Utilizando o lambda
    # Uma função lambda é declarada da seguinte forma
    # o argumento vem após do lambda
    # a expressão para ser retornada fica após o :
    quadrado_lambda = lambda x: x*x

    x_2_lambda = quadrado_lambda(x)
    print("x_2 : {0}, x_2_lambda {1}".format(x_2, x_2_lambda))

if __name__ == "__main__":
    main()
```

9. Plotando gráficos com o matplotlib

O modulo **matplotlib** é um dos mais utilizados no python para a plotagem de gráficos, esse módulo tem uma sintaxe semelhante a utilizada no **matlab**, para mudar a cor do gráfico, o simbolo utilizado , etc. Utilizando-se desse modulo juntamente com o modulo **numpy** é possível produzir gráficos de funções complexas em muitas aplicações. Na área acadêmica o uso do python está cada vez mais presente, a exemplo disso tem-se o caso recente da “foto” de um buraco negro, no qual foi utilizado o python juntamente dos modulos **numpy** e **matplotlib** para processar os dados e plotar a imagem.

9.1. Plotagem básica

Para exemplificar o uso do numpy com o matplotlib foi escolhido a plotagem de uma função seno. Seguindo as equações abaixo:

$$y = sen(2\pi * x) \quad , \quad x = [0.1, 0.2, 0.3 ..., 2] \qquad (1)$$

```
import numpy as np
import matplotlib.pyplot as plt

# Definindo o eixo x e a função y
x = np.arange(0,2,0.01)
y = np.sin(2*np.pi*x)

plt.plot(x,y)
plt.show()
```

Nota-se que este exemplo acima é um dos usos mais simples possíveis da biblioteca, nos exemplos seguintes serão abordados como definir e salvar figuras, múltiplos plots e outras configurações da plotagem com a **matplotlib**.

9.2. Plotagem com mais opções

Como falado anteriormente o matplotlib aceita um leque de opções para plots, como adicionar *grid*, mudar a cor dos plots, adicionar várias funções no mesmo plot, adicionar legendas nas funções e nos eixos além de muitas outras opções. Para exemplificar as opções apresentadas o codigo abaixo foi produzido:

```
import numpy as np
import matplotlib.pyplot as plt

# Definindo o eixo x ,função y e z
x = np.arange(0, 2, 0.01)
y = np.sin(2*np.pi*x)
z = np.cos(2*np.pi*x)

plt.plot(x, y, 'r', label="seno")
plt.plot(x, z, 'b', label="coseno")  # Adicionando uma segunda função
plt.legend()                         # Aplicando as legendas das funções
plt.grid('on')                       # Adicionando grid ao plot
plt.xlabel("tempo")                  # Adicionando uma legenda no eixo x
plt.ylabel("volts")                  # Adicionando uma legenda no eixo y
plt.show()                           # mostrando o plot
```

Há também a opção de adicionar vários subplots a mesma figura, para isso é utilizado o *plt.subplot(Nºlinhas,Nºcolunas,Nºindex)*, os numéros de colunas e linhas delimita quantos gráficos e como os gráficos estarão dispostos, o Nº do index basicamente altera a ordem dos gráficos.

```
      # Importando as bibliotecas necessárias
import numpy as np
import matplotlib.pyplot as plt

# Definindo o eixo x ,função y e z
x = np.arange(0, 2, 0.01)
y = np.sin(2*np.pi*x)
z = np.cos(2*np.pi*x)
```

```
plt.subplot(211)                    # Registrando o primeiro subplot
plt.plot(x, y, 'r', label="seno")
plt.legend()                        # Aplicando as legendas das funções
plt.grid('on')                      # Adicionando grid ao plot
plt.xlabel("tempo")                 # Adicionando uma legenda no eixo x
plt.ylabel("volts")                 # Adicionando uma legenda no eixo y
plt.subplot(212)                    # Registrando o segundo subplot
plt.plot(x, z, 'b', label="coseno") # Adicionando uma segunda função
plt.legend()                        # Aplicando as legendas das funções
plt.grid('on')                      # Adicionando grid ao plot
plt.xlabel("tempo")                 # Adicionando uma legenda no eixo x
plt.ylabel("volts")                 # Adicionando uma legenda no eixo y
plt.savefig('plot.png')             # Salvando a figura
```

10. Exercícios com funções e plots

Abaixo estarão 2 exercícios para serem executados utilizando o conhecimento de funções e de estruturas de repetição e decisão para consolidar os conhecimentos apresentados na apostila até este momento.

10.1. Função temporal da voltagem de um capacitor em um circuito de RC série

Exercício: Escreva um código que execute as instruções abaixo, utilizando somente o que foi exposto nos códigos anteriores e os dados:

Função exponencial e^x com o numpy $\rightarrow$ np.exp()

Função da voltagem do capacitor no circuito RC:

$$Vc = Vin * e^{-\tau} \quad , \quad \tau = \frac{-t}{(R * C)} \tag{1}$$

Dados de entrada: $Vin = 5$, $R = 5$, $C = 0.1$, $t = [0.1, 0.2, 0.3 \ldots, 3]$

Com base nos dados apresentados faça os passos abaixo:

- Declare uma função que receba os argumentos de t, Vin, R e C e retorne Vc
- Na função main() aliemente a função de Vc e plote os dados de Vc versus tempo.

10.2. Achar o mínimo e máximo de uma função

Exercício: Escreva um código que execute as instruções abaixo, utilizando somente o que foi exposto nos códigos anteriores :

- Elabore um codigo dividido em função principal e funções que serão chamadas pela principal
- Elabore uma função que receba os dados de um vetor e ache o menor valor dos dados recebidos e retorne esse valor juntamente com o valor do index de y que tem o valor mínimo
- Elabore uma função que ache o maior dos valores de um vetor e retorne esse valor juntamente com o com o valor do index
- Elabore uma função matemática de sua escolha, ex (x^2 , sin(x), ...) que receba dados de um vetor x de entrada(representando dados contínuos)
- Na função principal declare o vetor de entrada x, chame a função matemática e guarde o valor em uma variável chamada y. Com y alimente as funções de mínimo e máximo e guarde como mim_v e max_v respectivamente.
- Após os passos acima plote a função y juntamente com os pontos de máximo e mínimo
- Nota-se que para poder plotar 1 ponto só em um gráfico é necessário a troca do simbolo, para isso coloque : ‘r^’ como um dos argumentos na função plt.plot(), dessa forma o plot irá mudar para cor vermelha com um simbolo de triangulo a cada ponto do gráfico

10.3. Resolução exercicio Vc :

```
import numpy as np
import matplotlib.pyplot as plt

def Vc_RC(t, r=5, c=0.1, vin=1):
    """
    Tensão de um capacitor em um circuito RC
    """
    tau = -t/(r*c)
    vc  = vin*(1 - np.exp(tau))
    return vc

def main():
    t  = np.arange(0, 3, 0.1)
    vc = Vc_RC(t)
    plt.plot(t, vc, 'b', label="tensão VC calculada")
    plt.legend()
    plt.grid()
    plt.xlabel("tempo")
    plt.ylabel("tensão")
    plt.show()

if __name__ == "__main__":
    main()
```

10.4. Resolução exercicio de mínimo e máximo :

```
import matplotlib.pyplot as plt
import numpy as np
def min_func(y):
    id  = 0
    min_v = y[0]
    min_id = id
    while id + 1 < len(y):
        if y[id] < min_v:
            min_v = y[id]
            min_id = id
        id = id + 1
    return min_id, min_v
def max_func(y):
    id  = 0
    max_v = y[0]
    max_id = id
    while id + 1 < len(y):
        if y[id] > max_v:
            max_v = y[id]
            max_id = id
        id = id + 1
    return max_id, max_v
def main():
    t  = np.arange(0, 0.5, 0.01)
    y = np.sin(2*np.pi*t)
    min_id, min_v = min_func(y)
    max_id, max_v = max_func(y)
    plt.plot(t, y, 'b', label="seno")
    plt.plot(t[min_id], min_v, 'r^', label="ponto mínimo")
    plt.plot(t[max_id], max_v, 'g^', label="ponto máximo")
    plt.legend()
    plt.show()
if __name__ == "__main__": main()
```

11. Classes e Objetos

Nessa seção os conceitos de classes e objetos serão apresentados de maneira simplificada e pratica, visto que a programação orientada a objetos (POO) é um tema relativamente extenso.

Objetos são agrupamentos de variáveis e funções em uma entidade. Esses objetos são criados a partir de um modelo definido por uma classe. Digamos que quiséssemos uma classe para definir objetos do tipo ponto, com os tributos x e y:

```
class ponto :
x = None
y = None

ponto1 = ponto()
ponto1.x = 1
ponto1.y = 2

print(ponto1.x)
print(ponto1.y)
```

Essa maneira de definir classes pode parecer confortável em um primeiro momento, porem quando pensamos que pode ser necessário trabalhar com classe que contenham muitos atributo ou atributos que não necessariamente sejam conhecidos, percebemos a necessidade de definirmos uma função de inicialização da classe, dessa maneira o programa ficaria:

```
class ponto: #Criando classe ponto
def __init__(self,x,y): #Definindo função de inicialização
      self.x = x #Atributos da classe recebendo os valores das entradas
        self.y = y

ponto1 = ponto(1, 2) #Criando o objeto ponto1 da classe ponto
print(ponto1.x)
print(ponto1.y)
```

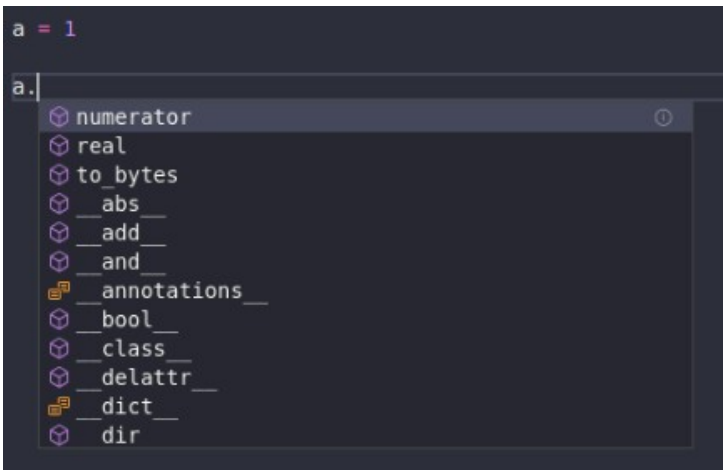
Assim como atributos, também é possível definir funções em uma classe :

```
from math import atan, degrees

class ponto: #Criando classe ponto
def __init__(self,x,y): #Definindo função de inicialização
        self.x = x #Atributos da classe recebendo os valores das entradas
        self.y = y

def getPolar(self):
        mod = (self.x**2 + self.y**2)**0.5
        ang = degrees(atan(self.y/self.x))
        return mod,ang

ponto1 = ponto(1, 1) #Criando o objeto ponto1 da classe ponto

print(ponto1.x)
print(ponto1.y)
print(ponto1.getPolar())
```

Nesse ponto é interessante fazer a seguinte pergunta:

- **Tem classes pré-definidas e quais são essas classes ?**
- **Resposta curta**: sim, basicamente tudo

Tudo o que fizemos envolveu classes e objetos, de variáveis do tipo *int* até dicionários são objetos no Python. Podemos constatar isso visualizando seus métodos :



REFERENCIAS BIBLIOGRÁFICAS