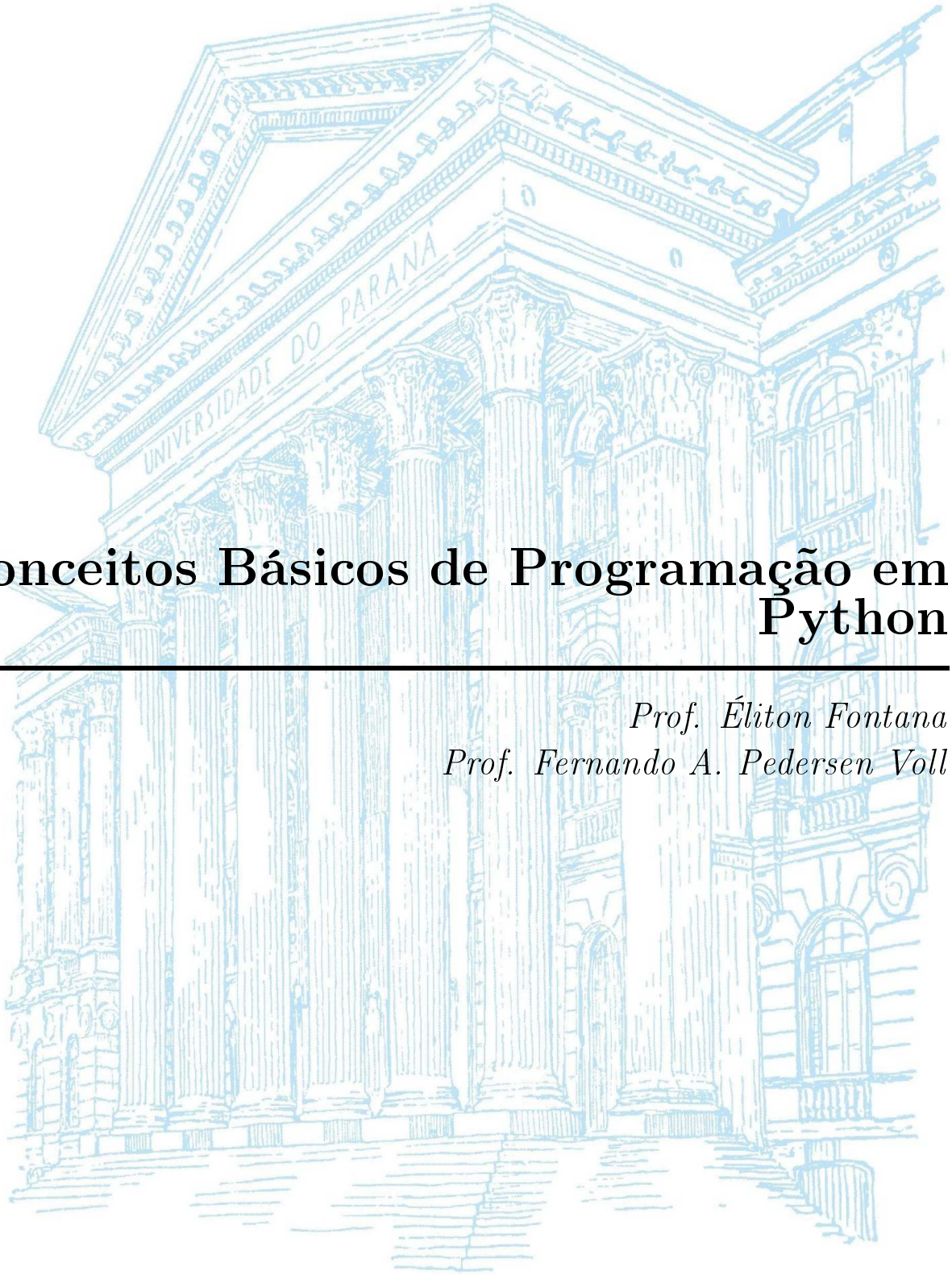
Universidade Federal do Paraná – UFPR
Departamento de Engenharia Química



Conceitos Básicos de Programação em Python

Prof. Éliton Fontana
Prof. Fernando A. Pedersen Voll

2020/1

Conteúdo

1 Introdução

Python é uma linguagem de alto nível amplamente utilizada pra computação científica. Existem diversos ambientes de desenvolvimento integrados (IDE's) disponíveis que permitem construir e compilar algoritmos desenvolvidos em Python. Recomenda-se a instalação do pacote *Anaconda 3.7*, pois este já possui várias bibliotecas científicas pré-instaladas, além de possuir o IDE *Spyder*. Os arquivos de instalação podem ser encontrados em `https://www.anaconda.com/distribution/`.

Uma das principais vantagens em utilizar Python é o grande número de bibliotecas disponíveis para os mais diversos fins, como por exemplo:

- NumPy;
- SciPy;
- Pandas;
- sklearn;
- matplotlib;
- math;

Em Python, comentários são adicionais usando # no início da linha. Comentários em blocos, ocupando várias linhas, podem ser definidos entre 3 aspas duplas.

2 Tipos de Dados

As variáveis utilizadas podem possuir diferentes tipos, dependendo de como forem definidas ou calculadas. O tipo de cada variável pode ser observado no explorador de variáveis. Os principais são os seguintes:

- **str**: variável do tipo string, usada para texto;
- **int**: variável do tipo inteiro;
- **float**: variável do tipo real (ponto flutuante);
- **list**: sequência de variáveis, onde cada elemento pode ser manipulado separadamente. Definida usando [ ];
- **tuple**: também representa uma sequência de variáveis, porém após definida não pode ter seu elementos alterados. Definida usando ( );

Mais detalhes sobre como operar estes tipos de variáveis são apresentados a seguir.

2.1 Matrizes e Vetores

Matrizes e vetores não são considerados como *tipos* de variáveis distintos, sendo a diferença para as demais variáveis somente sua dimensão. Não existe uma função dedicada para operar com matrizes em Python, por isso é altamente recomendável utilizar a biblioteca NumPy. Para isso, deve-se definir no início do código:

```
import numpy as np
```

Esta biblioteca já está previamente instalada no Anaconda, por isso basta importá-la. O nome *np* é utilizado como um 'apelido' para a biblioteca, facilitando com que funções

específicas sejam chamadas ao longo do código. Por exemplo, para definir um vetor, pode-se utilizar a função *array* disponível no NumPy, como no seguinte exemplo:

```
A = np.array([1, 1.5, 2, 2.5])
```

Isto irá gerar um vetor linha A com quatro posições. Os argumentos de entrada das funções são definidos entre parânteses. No caso desta função em particular, este argumento deve ser uma lista de valores.

Para definir uma matriz, pode-se utilizar a mesma função, porém o argumento de entrada deve ser uma lista com a dimensão correspondente. Por exemplo, considere uma matriz 2×3:

```
B = np.array( [ [1, 2, 3], [3, 2, 1] ] )
```

Observe que o argumento de entrada continua sendo uma única lista (agrupados dentro de colchetes), porém os elementos desta lista também são listas, neste caso cada uma contendo 3 elementos.

Os elementos individuais de cada matriz (ou vetor) podem ser acessados indicando a posição correspondente entre colchetes ao lado do nome da matriz. ***Em Python, os elementos começam no índice 0***. Por exemplo, para acessar o elemento na primeira linha e na terceira coluna da matriz B, utiliza-se o comando $B[0, 2]$. Neste caso, isso irá retornar o valor 3.

Outra função do NumPy muito útil em cálculo numérico é a *linspace*, que permite criar um vetor com N elementos igualmente espaçados. Por exemplo, considere que seja necessário criar um vetor A que vai de 0 até 2 contendo 15 elementos igualmente espaçados. Isto pode ser feito com o comando:

```
A = np.linspace(0,2,15)
```

Esta função necessita três argumentos de entrada, sendo o primeiro o valor inicial, o segundo o final e o terceiro o número e elementos.

2.2 Listas, Tuplas e Dicionários

Listas, tuplas e dicionários são formas de agrupar vários elmentos em um único termo. Estes agrupadores podem conter qualquer tipo de elementos, diferente de vetores e matrizes.

Por exemplo, pode-se criar uma lista contendo como primeiro elemento uma matriz, como segundo elemento uma string, como terceiro elemento outra lista, etc.

A diferença básica entre estes agrupadores é a seguinte:

- **Lista:** Sequência *mutável* de elementos de qualquer tipo. Qualquer elemento pode ser alterado individualmente, por isso costumam ser a forma mais conveniente de agrupar valores. É definida agrupando os termos entre colchetes. Por exemplo, para pode-se criar uma lista de 3 valores como:

  lista_val = [1, 2, 3]

  Caso seja necessário alterar um destes valores, basta redefir a posição equivalente. Por exemplo, para fazer o segundo valor passar a ser 4, basta definir:

  lista_val[1] = 4

  Com isso, a lista passa a ser $[1, 4, 3]$.

- **Tuplas:** são semelhantes as listas, porém é uma sequência *imutável* de elementos, ou seja, após definida não pode mais ser alterada. Isto inclui adicionar novos elementos ou remover elementos individuais. De forma geral, tuplas são elementos mais simples e "seguros" do que listas, devendo ser utilizada quando a informação definida não pode correr o risco de ser alterada (*read-only mode*). A definição de tuplas é feita agrupando os itens entre parênteses, por exemplo:

  tupla_val = (1, 2, 3)

  Caso o usuário tente alterar o valor de algum elemento, como por exemplo:

  tupla_val[1] = 4

  Isto irá gerar uma mensagem de erro e a tupla não será alterada.

- **Dicionários:** Também são sequências de valores *mutáveis*, semelhante às listas, com a diferença que nos dicionários pode ser atribuído uma palavra-chave para identificar cada elemento da sequência, facilitando seu acesso posteriormente. Os dicionários são

definidos entre chaves, sendo cada elemento definido da forma *palavra-chave : elemento*, por exemplo:

dict_val = { 'primeiro' : 1, 'segundo' : 2, 'terceiro': 3 }

Para acessar os valores, utiliza-se a palavra-chave como argumento. Por exemplo, *dict_val['segundo']* irá retornar o valor 2.

Comandos Úteis para Listas

Como as listas podem ser alteradas, existem diversos comandos úteis que podem ser aplicados, valendo destacar os seguintes:

- list.**append**(x): adiciona o elemento x ao fim da lista *list*;
- list.**insert**(i,x): Adiciona o elemento x na posição i da lista *list*;
- list.**clear**(): remove todos os elementos da lista *list*;
- **len**(list): retorna o tamanho da lista *list*;

3 Operadores Lógicos

3.1 For

O comando **for** em Python opera de uma maneira um pouco diferente de outras linguagens, onde usualmente este comando avança o índice de um contador numérico. Em Python, o comando for é utilizado para iterar ao longo de uma sequência, que pode ser uma lista, uma tupla, um dicionário, uma string, entre outros. Considere o seguinte exemplo:

```
1 #Definindo uma lista
2 A = ['a', 'b', 'c']
3
4 #Imprimindo os elementos da lista
5 for x in A:
6     print(x)
```

Observe que a variável x não precisa ser previamente definida. Além disso, é importante destacar que em Python ***espaços em branco no início da linha são importantes!.*** A identação é utilizada para definir blocos ao longo do programa, ou seja, funciona de maneira similar a um *begin/end* em outras linguagens. No exemplo anterior, a linha 6 faz parte do bloco definido pelo for.

Na implementação de métodos numéricos, é muito comum ser necessário avançar ao longo de uma quantidade finita de elementos de um contador. Para isso pode-se utilizar a função *range( )*. Esta função opera com 3 argumentos de entrada, sendo o primeiro o valor inicial, o segundo o valor final e o terceiro o passo desejado. Caso o último valor não seja informado, é utilizado o passo padrão de 1. Além disso, ***o último valor não é utilizado no bloco***. Considere os seguintes exemplos:

```
1 #Imprimindo valores ímpares de 1 a 9
2 for x in range(1,10,2):
3     print(x)
4
5 #Imprimindo todos os valores de 1 a 9
6 for x in range(1,10):
7     print(x)
```

Observe que no segundo exemplo o valor 10 não é imprimido.

3.2 While

O comando *while* é utilizado de maneira similar à outras linguagens, repetindo o loop enquando uma dada condição se mantiver verdadeira. Considere o exemplo:

```
#Definindo valores
x = 1
y = 10
#Imprimindo x equanto for menor ou igual a y
while x <= y:
    print(x)
    x += 1 #Incremento
```

Neste caso, as variáveis utilizadas devem ser previamente definidas. O comando $x+=1$ incrementa em 1 a variável x, sendo equivalnete a $x = x + 1$.

3.3 If

O condicional *if* é definido de modo a checar se uma dada condição é satisfeita, então o bloco é executado. Para checar outra condição, utiliza-se o comando *elif*, que corresponde ao *else if* utilizado em outras linguagem. De modo semelhante, o comando *else* é executado caso nenhuma condição seja satisfeita, como no exemplo a seguir.

```
#Pedindo para o usuário entrar com um número inteiro
x = input('Digite um número inteiro entre 1 e 3')
#Convertendo o número de string para inteiro
a = int(x)
#Imprimindo informações sobre o número
if a == 1:
    print('número 1')
elif a == 2:
    print('número 2')
elif a == 3:
    print('número 3')
else:
    print('número fora do intervalo')
```

Neste exemplo, é utilizado a função *input()* para solicitar ao usuário que um valor seja informado. Esta função armazena o valor como string, sendo necessário converter para inteiro usando a função *int()*. Observe que para comparar o valor é utilizado ==. O comando = é usado para *definir* uma variável, enquanto que == é usado para checar a igualdade.

4 Criando e Utilizando Funções

Funções são rotinas que relacionam um conjunto de entradas com um conjunto de saídas, sendo executadas somente quando chamadas no código. Para definir uma função em Python, utiliza-se a sintaxe:

def nome_da_funcao(argumentos_de_entrada): bloco_de_execução

No bloco de execução define-se a rotina interna da função. Todo este bloco deve estar corretamente identado, ou seja, deve estar com uma tabulação de distância de onde o comando **def** foi definido. Quando aplicada em métodos numéricos, as funções costumam ser utilizadas para retornar valores com base em uma entrada. Para retornar algo específico como resultado da função, utiliza-se o comando **return** no bloco de execução.

Por exemplo, considere um função simples para calcular o volume de um gás ideal com base em uma temperatura e pressão conhecidas:

```
1 #Definindo o nome e argumentos de entrada
2 def vol_gas_ideal(P,T):
3     #Constante R usada somente na função
4     R = 8.314;
5     #Calculando o volume
6     V = R*T/P
7     #Pedindo para a função retornar o valor de V
8     return V
9
10 #Argumentos de entrada
11 P1 = 1E5
12 T1 = 300
13 #Usando a função
14 V1 = vol_gas_ideal(P1,T1)
```

É importante destacar que o número e o tipo dos argumentos de entrada da função devem ser exatamente iguais aos definidos quando a função é criada. Neste caso, 2 escalares. Outro ponto importante é que a variável R, definida no bloco de execução, só é reconhecida neste bloco, ou seja, caso essa variável for chamada fora da função ela não será reconhecida.

As funções são as ferramentas mais versáteis de um código, podendo receber diversos argumentos de entrada (incluindo outras funções) e realizar operações de forma simples e

organizada. Considere o seguinte exemplo para calcular o zero de uma função usado Newton-Raphson:

```
def newton(f,df,x0,prec,max_iteracoes):
    '''Função para calcular o zero de uma função f(x) usando Newton-Raphson
    - Argumentos de entada:
        f : função f(x)
        df : derivada da função f(x)
        x0 : chute inicial
        prec : precisão desejada
        max_iteracoes : número máximo de iterações

    - Argumento de saída:
        xn : valor de x tal que abs(f(x)) < prec'''

    #Inicializado a variável xn com o chute inicial
    xn = x0
    #contador do número de iterações
    cont = 0
    #Implementar até a precisão ser alcançada
    while f(xn) > prec:
        #checando se a derivada não é zero
        if df(xn) == 0:
            #Caso for, imprimir a mensagem de erro e parar o código
            print('Divisão por zero')
            break

        #Se a derivada não for nula, calcular a próxima iteração
        xn = xn - f(xn)/df(xn)

        #Incrementando o contador e parando se o número máximo for atingido
        cont += 1
        if cont > max_iteracoes:
            print('Número máximo de iterações')
            break
    #Se tudo ocorrer bem, retorna o valor encontrado
    return xn
```

Para utilizar esta função, é preciso definir outras duas funções, além dos parâmetros chute-inicial, precisão e número máximo de iterações, como mostrado no exemplo a seguir.

```
#Exemplo de utilização da função newton
import numpy as np

def fx(x):
    y = x**3 + x**2 + np.exp(x)
    return y

def dfx(x):
    y = 3*x**2 + 2**x + np.exp(x)
    return y

#chute inicial 0, precisão 0.001 e máximo 100 iterações
x0 = newton(fx,dfx,0,0.001,100)
```

4.1 Plotando Gráficos

Os pacotes listados no início deste material possuem um número enorme de funções pré-programadas, como por exemplo as funções *array* e *linspace* do NumPy, mencionadas anteriormente. Para plotar gráficos, pode-se utilizar o pacote matlibplot. Por exemplo, para plotar a função $y = 0.5x + 2\sin(x) + \exp(x/10)$, pode-se utilizar a seguinte estrutura:

```
#Importanto o numpy para usar funções matemáticas
import numpy as np
#Importanto a função pyplot do matplotlib
import matplotlib.pyplot as plt

#Criando um vetor x entre 0 e 5 com 100 componentes
x = np.linspace(0,5,100)
#Calculando a função
y = 0.5*x + 2*np.sin(x) + np.exp(x/5)
#Obs.: As funções seno e exp não são definidas por padrão,
#sendo necessário usar as funções do NumPy

#Plotando y em função de x
plt.plot(x,y)
#Adicionando título aos eixos
plt.xlabel('x')
plt.ylabel('y')
```