Universidade Estadual de Campinas

Minicurso Python3

Autor: João Henrique Faria

Conteúdo

1 Introdução

Este minicurso tem a intenção de ensinar o básico da programação em Python. Atualmente, existem duas principais versões de Python em uso: Python2 e Python3. As diferenças entre as duas versões fogem do escopo desse curso, mas uma ressalva importante a ser feita é: a versão Python3 **não é retrocompatível**, ou seja, programas escritos em Python2 podem não rodar em Python3. Isso se deve ao fato de que uma parte das funções que existiam em Python2 foram alteradas ou removidas na versão mais atual. Entretanto, a maioria das funções em Python3 já existiam em Python2 - portanto, é bem provável que códigos escritos na versão mais atual também rodem na mais antiga. Por isso utilizaremos a versão Python3 nesse minicurso.

1.1 Pré Requisitos - Linux

A maioria das distribuições Linux já vem por padrão com as duas versões do Python instaladas. Para utilizar, existem três principais maneiras, citadas a seguir.

1. direto à partir do terminal:

   Para utilizar Python direto do terminal, basta digitar o comando:

   ```
   $ python3
   ```

   Vale ressaltar que dependendo da versão da distribuição, apenas a palavra "python"já é suficiente para executar a versão mais atual.

2. escrevendo em um editor de texto, e utilizando o comando *python*:

   Para esse caso, basta criar um arquivo com a extensão "**.py**", e editá-lo com qualquer editor de texto (kate, sublime, emacs, vim...) - recomenda-se utilizar o **sublime**, pois ele é bastante simples e também define cores diferentes para cada estrutura em Python (e em outras linguagens também). Vale ressaltar que a extensão do arquivo em Linux é apenas para caráter organizacional - para identificar a linguagem que o código foi escrito, ou para definir um programa padrão para abrir cada tipo de arquivo, por exemplo).

   Tendo o código já escrito, para executá-lo, basta abrir um terminal (na mesma pasta onde o arquivo está) e digitar o seguinte comando (onde "nome_do_arquivo.py"é o nome do arquivo onde o código foi escrito):

   ```
   $ python3 nome_do_arquivo.py
   ```

3. escrevendo em um editor de texto, e utilizando *shebang*:

   Shebang é o nome que se dá à primeira linha do código, iniciada sempre com os caracteres "#!". Essa linha não é obrigatória, mas é bastante utilizada, pois a partir dela é possível determinar a linguagem na qual o código foi escrita, e executar qualquer código (independente da linguagem) de maneira universal. Para isso, basta adicionar a seguinte linha no início do código:

   ```
   #!/usr/bin/env python3
   codigo
   ```

   Em seguida, é necessário trocar as permisões do arquivo, possibilitando que ele seja executado. Para isso, se usa o seguinte comando no temrinal:

   ```
   $ chmod +x nome_do_arquivo.py
   ```

   E finalmente, para executar o arquivo, basta digitar no terminal:

   ```
   $ ./nome_do_arquivo.py
   ```

Em Python também existem bibliotecas, que são um compilado de funções já prontas, com finalidades diversas e específicas. Por padrão, o Python já instala uma quantidade razoável de bibliotecas, mas algumas mais específicas é necessário instalar manualmente. Para isso, utiliza-se o gerenciador de bibliotecas *pip3*. Para instalar uma nova biblioteca em Python, utiliza-se o seguinte comando no terminal:

```
$ pip3 install nome_da_biblioteca
```

2 Estrutura Básica

Para escrever códigos em Python, é necessário seguir algumas regras básicas. A estrutura básica é a seguinte:

```
Comando1
Comando2
Comando3
[...]
ComandoN
```

Nessa estrutura, cada comando é escrito separadamente em uma linha (é possível escrever mais de um comando por linha, separando-os por **';'** - entretanto, isso não é recomendado pois linhas muito longas acabam deixando o programa confuso e difícil de entender). Cada um dos comandos do código será executado em sequência, de cima para baixo, um de cada vez.

2.1 Variáveis

Em um programa, as variáveis são os nomes dados pelo programador a determinados valores. Em Python, diferentemente de algumas linguagens (como **C**, por exemplo), não é necessário declarar previamente as variáveis. Também existem algumas regras na hora de criar variáveis, sendo elas:

1. todas as variáveis **devem** começar com uma letra (podendo essa ler maiúscula ou minúscula) ou *underline* (**'_'**), elas **nunca** devem começar com um número;
2. podem conter letras maiúsculas e minúsculas, números e *underline*;
3. não podem conter caracteres especiais como **'{'**, **'('**, '+', **'-'**, **'*'**, '/', '\', **';'**, **'.'**, **','** e **'?'**;
4. letras maíusculas e minúsculas são **diferentes** (**'c'** e **'C'** podem ser declaradas simultaneamente, e podem assumir valores diferentes).

Em Python, para declarar uma variável, basta escrever o nome que se quer dar a variável, seguido de um '=' e o valor dessa variável, como no exemplo mostrado abaixo.

```
x = 10
y = 20 + 10
z = 30 - x

nome = "Maria"
```

2.2 Tipos

Assim como em outras linguagens de programação, as variáveis em Python possuem tipos. Os principais tipos serão discutidos na próxima seção. Vale ressaltar que diferentemente de algumas linguagens, as variáveis podem mudar constantemente de tipo – em **C**, por exemplo, o tipo da variável é fixado logo na declaração. Portanto, em Python, o código exemplo a seguir é valido.

```
x = 1
x = 2.3
x = 'c'
x = 'palavra'
x = 'frase com mais de uma palavra'
```

2.2.1 Principais Tipos

Os principais tipos em Python são os seguintes:

1. **int** ou inteiro: corresponde aos números inteiros;

```
x = 10
```

2. **float**: corresponde aos números reais;

```
x = 3.1415
```

3. **str** ou string: corresponde a textos;

```
x = "texto"
x = "texto com espaco"
```

4. **bool** ou booleana: armazena **True** e **False**;

```
x = True
y = False
```

5. **lista**: agrupa um conjunto de elementos;

```
x = [1, 2, 3, 4, 5]
```

6. **tupla**: semelhante à lista, porém é imutável;

```
x = (1, 2, 3, 4, 5)
```

7. **dicionário**: também agrupa elementos, mas de um jeito mais organizado, dando "nomes"a cada elemento.

```
x = {"nome": Maria, "idade": 20, "altura": 1.55}
```

A função **type()** pode ser utilizada para identificar o tipo de uma variável, como no exemplo abaixo.

```
x = 10
type(x)

================
>> <class 'int'>
```

2.2.2 Conversão de Tipos

Em Python, também é possível converter o tipo de algumas variáveis, dependendo do tipo atual e do tipo desejado. Por exemplo, uma variável do tipo **int** pode ser alterada para **float** ou **str** sem nenhum problema. Isso muitas vezes se faz necessário pois algumas operações só podem ser realizadas com tipos específicos, ou também entre tipos específicos. Um exemplo disso se encontra abaixo.

```
a = 2
b = '2'

c = a + int(b)
d = str(a) + b

================
>> 4
>> '22'
```

2.2.3 Mudança de Base

Em Python, valores numéricos são armazenados sempre da mesma maneira. Por causa disso, operações entre diferentes bases (**decimal**, **octal** e **hexadecimal**, por exemplo) podem ser realizadas sem nenhuma limitação. As funções **int()** (para decimal), **oct()** (para octal) e **hex()** (para hexadecimal) podem ser utilizadas para converter os valores, caso uma base específica seja desejada, como no exemplo abaixo.

```
a = 10      # 10 em decimal
b = 0o10    # 8 em decimal
c = 0x10    # 16 em decimal

hex(a)
int(b)
oct(c)

================
>> '0xa'
>> 8
>> '0o20'
```

Vale ressaltar que o retorno das funções **oct()** e **hex()** é do tipo **str**, portanto operações com esse retorno tem que serem feitas com muito cuidado.

2.3 Funções Básicas

Para a continuidade do curso, é necessário conhecer as funções descritas a seguir.

2.3.1 Função *print()*

De maneira literal, essa função é usada para imprimir informações no terminal. Essas informações podem ser um texto desejado pelo programador, ou até mesmo os valores das variáveis do programa, como nos exemplos abaixo.

```
a = 10
print("O valor de a eh", a)

print("Esse eh o final do codigo.")

================
>> O valor de a eh 10
>> Esse eh o final do codigo.
```

Como pode ser visto no exemplo acima, cada vez que a função **print()** é chamada, ela imprime os valores desejados em uma nova linha. Isso acontece porque por padrão a função adiciona o caractere **'\n'** ao final dos valores que são passados para ela. Para trocar esse caractere, basta passar um novo como parâmetro para a função, como mostra o exemplo abaixo.

```
print("Comeco da frase, ", end='')
print("e final da frase.")

================
>> Comeco da frase, e final da frase.
```

Além disso, um recurso muito importante para manipular **strings** é a função **format()**. Usar essa função é uma das maneiras de mesclar texto com os valores de variáveis. Um exemplo disso é mostrado abaixo. Para adicionar os valores de variáveis com a função **format()**, deve-se adicionar **{:letra}**, onde 'letra' pode ser 'd' para **int**, 'f' para **float**, e 's' para **string**.

```
altura = 1.50
peso   = 70

print("A pessoa tem altura {:f} metros e peso {:d} quilos.".format(altura, peso))
```

```
================
>> A pessoa tem altura 1.500000 metros e peso 70 quilos.
```

Como vemos no exemplo acima, o valor da 'altura' é impresso com várias casas decimais, apesar de termos declarado apenas uma. Para limitar o número de casas decimais com as quais um número é impresso, pode-se substituir o valor da variável no texto por **{:.'casas'f}**, onde **'casas'** é a quantidade de casas decimais desejadas, como mostra o exemplo abaixo.

```
altura = 1.50
peso   = 70

print("A pessoa tem altura {:.1f} metros e peso {:d} quilos.".format(altura, peso))

================
>> A pessoa tem altura 1.5 metros e peso 70 quilos.
```

2.3.2 Função *input()*

Para possibilitar que o usuário adicione valores ao programa, usa-se a função **input()**. Essa função faz com que o programa pare sua execução, e aguarde até que o usuário digite um valor no terminal. Esse valor então é armazenado na variável na qual a função foi associada. Vale ressaltar que chamadas subsequentes da função **input()** são executadas na ordem em que são declaradas no código. Um exemplo disso é mostrado abaixo. Além disso, por padrão, essa função retorna um valor do tipo **str**. Portanto, para valores numéricos, é preciso converte-los para o tipo desejado.

```
altura = float(input())
peso   = int(input())

print("A pessoa tem altura {:.1f} metros e peso {:d} quilos.".format(altura, peso))

================
>> 1.5
>> 70
>> A pessoa tem altura 1.5 metros e peso 70 quilos.
```

2.4 Expressões Aritméticas

Em Python, em adição às expressões básicas (soma, subtração, multiplicação e divisão) há também, já por padrão e sem a necessidade de utilizar bibliotecas específicas, algumas outras expressões. Elas se encontram explicitadas na próxima seção.

2.4.1 Principais Expressões

1. soma:

```
10 + 20

================
>> 30
```

2. subtração:

```
20 - 10

================
>> 10
```

3. multiplicação:

```
10 * 20
```

```
================
>> 200
```

4. divisão (com resultado real):

```
20 / 10

================
>> 2.0
```

5. divisão (com resultado inteiro ou truncado):

```
20 // 10
30 // 20

================
>> 2
>> 0
```

6. potência:

```
2 ** 10

================
>> 1024
```

7. resto da divisão (inteira):

```
30 % 20

================
>> 10
```

8. potências de 10:

```
2e2

================
>> 200.0
```

2.4.2 Precedência

As expressões matemáticas em Python possuem precedência, ou seja, a ordem na qual as operações são feitas pode ser diferente da ordem na qual as operações foram escritas. No exemplo abaixo, vemos um dos casos mais básicos de precedência, onde a multiplicação é realizada antes da soma.

```
10 + 20 * 2

================
>> 50
```

A ordem das expressões segue a sequência abaixo:

1. potência (**);
2. multiplicação (*) e divisão (/ e //), na ordem em que aparecem na expressão;
3. resto (%);
4. adição (+) e subtração (-), na ordem em que aparecem na expressão.

Para mudar a precedência das expressões, podemos usar parênteses, de modo que as expressões entre parênteses serão sempre executadas com prioridade, como no exemplo abaixo.

```
(10 + 20) * 2

================
>> 60
```

3 Expressões Relacionais

Expressões relacionais são comparações entre dois valores, que retornam um valor booleano (**True** ou **False**). As comparações que podem ser feitas são as seguintes:

1. igualdade (==):

```
10 == 20

================
>> False
```

2. diferença (!=):

```
10 != 20

================
>> True
```

3. maior que (>):

```
10 > 20

================
>> False
```

4. menor que (<):

```
10 < 20

================
>> True
```

5. maior ou igual que (>=):

```
10 >= 20
10 >= 10

================
>> False
>> True
```

6. menor ou igual que (<=):

```
10 <= 20
10 <= 20

================
>> True
>> True
```

4 Expressões Lógicas

Expressões lógicas são aquelas que realizam uma operação lógica (**and**, **or**,...) e assim como as relacionais, também retornam um valor booleano. As principais expressões lógicas estão exemplificadas a seguir.

1. **and**

   Tem retorno verdadeiro caso **todas** as comparações realizadas também sejam verdadeiras, como no exemplo abaixo.

```
(10 == 10) and (20 == 20) and (30 == 30)
(10 == 10) and (20 == 20) and (30 == 20)

================
>> True
>> False
```

2. **or**

   Tem retorno verdadeiro caso **alguma** das comparações realizadas seja verdadeira, como no exemplo abaixo.

```
(10 == 10) or (20 == 20) or (30 == 30)
(10 == 10) or (20 == 20) or (30 == 20)

================
>> True
>> True
```

3. **not**

   Tem retorno verdadeiro quando a comparação realizada tem retorno falso, e vice-versa.

```
not(10 == 10)
not(10 == 20)

================
>> False
>> True
```

5 Comandos Condicionais

Comandos condicionais são aqueles que permitem decidir se um determinado bloco de comandos deve ou não ser executado, à partir de uma expressão relacional ou lógica.

Em Python, um bloco de comandos é um conjunto de instruções. A linguagem requere que o início do bloco seja sempre com dois pontos **':'**, e que todas as instruções do bloco estejam identadas - essa identação pode ser com 'tab' ou 'espaços'.

O principal comando condicional é o **if**, que executa seu respectivo bloco caso sua expressão condicional seja **verdadeira**. Um exemplo é mostrado no código a seguir.

```
a = 2

if((a%2) == 0):
    print("O numero eh par.")

================
>> O numero eh par.
```

Variações do comando **if** são os comandos **else** e **elif**. Esses dois comandos são um complemento ao **if**, tornando possível usar a resposta dos testes anteriores para decidir se um bloco deve ou são ser executado. Por exemplo, no código abaixo, os testes serão realizados em sequência. Caso a condição do comando **if** seja verdadeira, seu bloco será executado e as demais partes serão ignoradas. Caso essa condição seja falsa, será testada a condição do comando **elif**, cujo bloco será executado caso sua condição seja verdadeira. O bloco referente ao comando **else** será executado **apenas** se **todas** os testes anteriores forem falsos.

```
a = 1

if(a > 0):
    print("O numero eh positivo.")
elif(a < 0):
    print("O numero eh negativo.")
else:
    print("O numero eh zero.")
```

```
================
>> O numero eh positivo.
```

Vale ressaltar que cada conjunto de condicionais **if** / **elif** / **else** deve sempre ser iniciado com um **if**. Após esse **if**, podem ser adicionados quantos **elif** forem necessários. O comando **else** no final é opcional, mas se usado, deve sempre ser colocado ao final do conjunto.

6 Listas

Listas são elementos que possibilitam armazenar diversos dados de maneira simplificada, não necessariamente de um mesmo tipo. Por exemplo, uma lista possibilita 'n' números, sem a necessidade de criar 'n' variáveis.

O acesso ao conteúdo da lista é feito através do **índice**, que é um valor numérico referente à posição do item desejado. Vale ressaltar que as listas podem ser modificadas - ou seja, pode-se tanto alterar os valores relacionados a cada posição, ou também a quantidade de itens na lista (para mais e também para menos).

A estrutura da lista é a seguinte:

```
lista = [valor_1, valor_2, valor_3, ..., valor_n]
```

Exemplos de listas estão explicitados abaixo.

```
lista_1 = [0, 1, 2, 3] # lista com numeros
lista_2 = [0, "string", 4.0] # lista com diferentes tipos
lista_3 = [] # lista vazia
```

Considerando uma lista de tamanho 'n', os valores válidos para o **índice** estão entre 0 e '(n-1)', pois o primeiro elemento da lista (da esquerda para a direita) recebe índice 0. Valores negativos para o índice também são válidos, e servem para referenciar a lista da direita para a esquerda (o famoso "de trás para frente"). Um exemplo do uso do índice é mostrado abaixo. Vale ressaltar que tentar acessar uma posição inexistente da lista faz com que o código retorne um erro, e seja interrompido.

```
lista = [0, 1, 2, 3]

a = lista[0]
lista[1] = lista[1] + 10
b = lista[1]
c = lista[-1]

print(a, b, c)

================
>> 0 11 3
```

Os principais recursos das listas estão descritos abaixo.

1. *slicing*:

   Slicing é criar uma nova lista apenas com os valores de um determinado intervalo de uma lista já existente, como no exemplo abaixo.

   ```
   lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

   nova_lista = lista[4:8]

   print(nova_lista)

   ================
   >> [4, 5, 6, 7]
   ```

2. função *len()*:

   Essa função retorna o tamanho da lista; é bastante útil na parte seguinte, envolvendo repetição, para evitar que os valores de índice fora do intervalo permitido.

```
lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

tamanho_lista = len(lista)

print(tamanho_lista)

================
>> 10
```

3. método *append()*:

   Adiciona itens ao final da lista. Formalmente, a estrutura contendo a lista, seguida de um ponto e uma 'função' é na realidade chamada método.

```
lista = [0, 1, 2, 3]

lista.append(4)

print(lista)

================
>> [0, 1, 2, 3, 4]
```

4. concatenação:

   Concatenar listas é juntar duas ou mais listas em uma só. No exemplo abaixo, vemos duas maneiras de fazer isso. Lembrando que na multiplicação, o valor que multiplica a lista é a quantidade de vezes que a lista será repetida.

```
lista_1 = [0, 1, 2]
lista_2 = lista_1 + lista_1
lista_3 = 4 * lista_1

print(lista_2)
print(lista_3)

================
>> [0, 1, 2, 0, 1, 2]
>> [0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2]
```

5. método *insert()*:

   Adiciona um elemento em uma posição específica da lista.

```
lista = [0, 1, 2]

lista.insert(1,4) # posicao, valor

print(lista)

================
>> [0, 4, 1, 2]
```

6. função *del()* e método *remove()*:

   Duas maneiras de remover itens da lista. Na primeira, se apaga o valor em uma posição específica; já na segunda, apaga-se um valor específico na lista - caso o valor apareça mais de uma vez, o valor com menor índice é apagado.

```
lista = [0, 1, 2, 3, 4, 5, 6]

del(lista[2])
lista.remove(5)

print(lista)

================
>> [0, 1, 3, 4, 6]
```

7 Comandos Repetitivos

Comandos repetitivos, como o próprio nome já diz, são comandos utilizados para repetir determinados blocos de comandos 'n' vezes, sem a necessidade de escrever 'n' vezes esse bloco. De maneira análoga aos comandos condicionais, os comandos repetitivos vão repetir seu respectivo bloco de comandos **enquanto** sua condição continuar verdadeira - o teste de condição é realizado sempre ao final do bloco.

7.1 Comando *while*

O comando **while** segue a seguinte estrutura:

```
while(condicao):
    comando(s)
```

Vale ressaltar que a condição é uma parte muito importante no **while**. Se logo no início essa condição for falsa, o programa nunca executa esse bloco de comandos. Caso a condição seja sempre verdadeira, cria-se um *loop* infinito - e o programa nunca encerra. Por isso, ao utilizar o **while**, deve-se garantir que sua condição se tornará falsa durante sua execução. Um exemplo é dado abaixo.

```
i = 1
while(i < 5):
    # numeros pares
    if((i%2) == 0):
        print(i)
    i += 1

================
>> 2
>> 4
```

7.2 Comando *for*

O comando **for** segue a seguinte estrutura:

```
for variável in lista:
    comando(s)
```

Para cada um dos elementos da lista, é realizado o bloco de comando, da seguinte maneira: se a lista ainda não acabou, o valor do próximo elemento é atribuído à **variável**, o bloco de comando é executado com esse valor, e o código retorna ao passo anterior, até atingir o final da lista.

```
lista = [2, 4, 6, 8, 10, 12]

for item in lista:
    print(item, end=' ')

================
>> 2 4 6 8 10 12
```

Quando se quer apenas repetir um determinado bloco de comandos por um número de vezes, não necessariamente dependendo do valor da lista atribuída ao *for*, pode-se utilizar a função *range()*. Com essa função, é possível criar uma lista numérica ordenada, com um determinado valor inicial, valor máximo acrescido de um e também valor de passo (variação de um item da lista para o próximo). Essa função é utilizada como descrito abaixo.

```
# valor inicial, valor maximo+1, e passo, respectivamente
for i in range(1, 11, 1):
    print(i, end=' ')

================
>> 1 2 3 4 5 6 7 8 9 10
```

7.3 break e continue

O comando **break** faz com que o laço realizado seja interompido, passando a executar o próximo comando após o final do bloco de comandos executado pelo laço. Um exemplo é mostrado abaixo.

```
for i in range(1, 11, 1):
    print(i, end=' ')

    if(i == 6):
        break

================
>> 1 2 3 4 5 6
```

O comando **continue** faz com que o laço "pule"para o final, como no exemplo abaixo.

```
for i in range(1, 11, 1):
    if((i%2) != 0):
        continue
    print(i, end=' ')

================
>> 2 4 6 8 10
```

8 Strings

Strings em Python são listas imútáveis de caracteres, e são representadas por uma sequência de caracteres entre aspas simples (') ou duplas ("), como abaixo.

```
string = "string em Python"
print(string)

================
>> string em Python
```

Por ser uma lista, algumas das funções e métodos citadas acima funcionam perfeitamente com a *string*. Um exemplo é mostrado abaixo. Entretando, diferentemente da lista, a *string* é imutável, ou seja, uma vez declarada, não é possível alterar o valor de seus elementos - o programa é encerrado e retorna um erro. Entretanto, é possível redeclarar a *string*, mundando portanto o valor da variável.

```
string = "string em Python"
print(string[3:6])
print(2*string)
print(string + " (:")

string = "banana"
print(string)
```

```
================
>> ing
>> string em Pythonstring em Python
>> string em Python (:
>> banana
```

Também existem algumas funções e métodos exclusivos para a *string*, explicitados abaixo.

1. método *strip()*:

   Retorna uma *string* sem os espaços em branco e também sem as quebras de linha no começo e no final da string, como no exemplo abaixo.

   ```
   string = "          \nisso eh uma \nstring                  \n"

   print(string.strip())

   ================
   >> isso eh uma
   >> string
   ```

2. operador *in*:

   O operador *in* verifica se uma *substring* faz parte de uma *string*, retornando verdadeiro para o caso onde a *string* contém a *substring* e falso para quando não contém, como no exemplo abaixo.

   ```
   string = "isso eh uma string"

   a = "eh" in string
   b = "duas" in string

   print(a, b)

   ================
   >> True False
   ```

3. método *find()*:

   Esse método retorna onde (em qual índice) uma determinada *substring* começa na *string*. Vale ressaltar que caso a *string* não contenha a *substring*, o valor de retorno é '-1', como no exemplo abaixo.

   ```
   string = "isso eh uma string"

   a = string.find("eh")
   b = string.find("duas")

   print(a, b)

   ================
   >> 5 -1
   ```

4. método *split()*:

   O método *split()* separa uma determinada *string* nos lugares onde aparece o caractere indicado - um exemplo é mostrado abaixo. Se nenhum caractere for indicado, por padrão o método separa a *string* nos lugares onde aparecem **'tab'** e **'\n'**. É importante lembrar também que esse método retorna uma lista com as *substrings* resultantes, e que essas *substrings* podem ser vazias.

```
# espacamento com 'tab', 'espaco' e 'espaco' duplo, respectivamente
string = "isso	eh uma  string"

print(string.split(" "))
print(string.split())

================
>> ['isso\teh', 'uma', '', 'string']
>> ['isso', 'eh', 'uma', 'string']
```

5. método *replace()*:

   O método *replace()* substitui a ocorrência de um caractere ou *substring* na *string* indicada. Vale ressaltar que como a *string* é **imutável**, na realidade esse método cria uma nova *string* com as substituições, mantendo a original intacta - como no exemplo abaixo.

```
string1 = "isso eh uma string"

string2 = string1.replace('uma', 'outra')

print(string1)
print(string2)

================
>> isso eh uma string
>> isso eh outra string
```

Ainda importantes, existe a função *list()* e o método *join()*. A primeira é usada para converter uma *string* em uma lista com os caracteres da mesma. Já o segundo é justamente o contrário, e transforma uma lista ou sequência em uma *string*, concatenando os elementos desta lista. Um exemplo para ambos os casos é mostrado abaixo.

```
string1 = "uma string"

lista = list(string1)

string2 = ''.join(lista)

print(lista)
print(string2)

================
>> ['u', 'm', 'a', ' ', 's', 't', 'r', 'i', 'n', 'g']
>> uma string
```

9 Tuplas

Tuplas são semelhantes às listas, ou seja, são sequências de dados (podendo até ser de diferentes tipos). Entretanto, diferentemente das listas, tuplas são **imutáveis**. A principal vantagem de utilizar tuplas no lugar de listas é que por ser imutável, as tuplas são melhores tanto quanto em tempo de processamento quanto em alocação de memória - a grosso modo, utilizar tuplas no lugar de listas (quando possível) faz com que o programa seja mais eficiente (e elegante). Um exemplo é mostrado abaixo.

```
semana = ('Domingo', 'Segunda', 'Terca', 'Quarta', 'Quinta', 'Sexta', 'Sabado')

print(semana)

================
>> ('Domingo', 'Segunda', 'Terca', 'Quarta', 'Quinta', 'Sexta', 'Sabado')
```

Tuplas também possuem recursos chamados **empacotamento** e **desempacotamento**. O primeiro consiste em criar uma tupla implicitamente, apenas separando seus valores por vírgula. Já o segundo consiste no oposto: dividir os valores de uma tupla em diferentes variáveis. Um exemplo de ambos é mostrado abaixo.

```
tupla = 10, 20, 30, 40

print(tupla)

a, b, c, d = tupla

print(a, b, c, d)

================
>> (10, 20, 30, 40)
>> 10 20 30 40
```

10 Dicionários

Dicionários são estruturas que associam uma chave a um valor. De maneira simplificada, dicionários são semelhantes às listas, mas para cada item, ao invés de um índice, é associado um "nome". Os itens do dicionário podem ser de tipo qualquer; entretanto, as chaves precisam ser imutáveis e únicas. Um exemplo é mostrado abaixo.

```
dic = {"Joao": 134567, "Maria": 165478, "Gustavo": 143768}

print(dic["Joao"])

================
>> 134567
```

Vale ressaltar que apesar de a estrutura do dicionário ser declarada em uma ordem específica, não necessariamente o dicionário será salvo nessa ordem. Por exemplo, no código acima, foram declarados em sequências os RAs de João, Maria e Gustavo. Se imprimirmos a variável que leva o dicionário, é possível que a ordem não seja a mesma. Além disso, é possível também adicionar novos elementos no dicionário, como abaixo.

```
dic = {"Joao": 134567, "Maria": 165478, "Gustavo": 143768}

dic["Ricardo"] = 160094

print(dic)

================
>> {'Joao': 134567, 'Maria': 165478, 'Gustavo': 143768, 'Ricardo': 160094}
```

Assim como nas *strings*, o operador **in** verifica se uma determinada chave está no dicionário, retornando verdadeiro para o caso positivo e falso para o negativo. Além dele, o método *items()* retorna tuplas dos pares chave/valor. Ambos estão explicitados abaixo.

```
dic = {"Joao": 134567, "Maria": 165478, "Gustavo": 143768}

print("Joao" in dic)

print(dic.items())

================
>> True
>> dict_items([('Joao', 134567), ('Maria', 165478), ('Gustavo', 143768)])
```

11 Funções

Um ponto muito importante em programação é a organização. Portanto, quebrar um código grande em várias partes menores é essencial para criar um bom programa. Para fazer essa modularização, criamos as funções, que são trechos de código declarados separadamente da rotina principal.

Funções podem ser criadas por dois principais motivos: o primeiro, criar um trecho de código que tem uma função específica, e que pode ser aproveitada em diferentes projetos; segundo, evitar repetições desnecessárias de comandos - criar uma função, e chamá-la repetidas vezes é bem melhor organizado do que repetir diversas vezes o mesmo bloco de código.

11.1 Estrutura Básica

A estrutura básica de uma função é explicitada a seguir. Os **parâmetros** são variáveis, inicializadas com os valores indicados na invocação da função. Além disso, funções podem ou não ter um **valor de retorno**, dependendo da maneira com a qual forem implementadas.

```
def nome_da_funcao(parametro1, parametro2, ..., parametroN):
    comando(s)

    return valor_de_retorno
```

Ao invocar a função, deve-se passar todos os parâmetros necessários para que ela seja executada - esses parâmetros podem ser tanto valores declarados diretamente, quanto variáveis previamente utilizadas durante o código. Um exemplo é mostrado abaixo. Além disso, é necessário também declarar a função antes de sua utilização no código, caso contrário o programa deixará de ser executado e retornará um erro.

```
def faz_conta(num1, num2, operador):
        if(operador == '+'):
                resp = num1 + num2
                return resp
        elif(operador == '-'):
                resp = num1 - num2
                return resp
        else:
                print("Operador não suportado!")
                return
def printa_conta(num1, num2, operador):
        if(operador == '+'):
                print(num1 + num2)
        elif(operador == '-'):
                print(num1 - num2)
        else:
                print("Operador não suportado!")

valor = faz_conta(10, 20, '+')
print(valor)
valor = faz_conta(10, 20, 'b')
print(valor, type(valor))
printa_conta(10, 20, '+')

================
>> 30
>> Operador não suportado!
>> None <class 'NoneType'>
>> 30
```

11.2 Argumentos nomeados

Até agora, na invocação da função, era necessário passar os argumentos na mesma ordem na qual foram declarados. Entretanto, é possível variar essa ordem, desde que se atribua o valor desejado à variavel durante a invocação. Além disso, é possível também chamar a função com menos parâmetros do que ela necessita, desde que durante a declaração da função valores *default* sejam atribuídos às variáveis não utilizadas na invocação. Um exemplo é mostrado abaixo.

```
def printa_div(num1, num2=1):
    print(num1 / num2)

printa_div(20)
printa_div(num2=10, num1=100)

================
>> 20.0
>> 10.0
```

12 Matrizes

Matrizes são generalizações de vetores (ou listas) já discutidos anteriormente. A grosso modo, uma matriz nada mais é do que uma "lista de listas", ou seja, cada elemento da lista principal é uma outra lista. Exemplos de declaração de matrizes estão explicitados abaixo.

```
matriz1 = [[], [], []]
matriz2 = [[] for i in range(3)]

print(matriz1)
print(matriz2)

================
>> [[], [], []]
>> [[], [], []]
```

Para acessar um elemento específico de uma matriz, fazemos como no exemplo abaixo. Vale ressaltar que tentar acessar posições inválidas na matriz (tanto para as linhas quanto para as colunas) faz com que o código retorne um erro.

```
matriz = [[1, 2], [3, 4]]

ind_lin = 0

while(ind_lin < len(matriz)):
        ind_col = 0

        while(ind_col < len(matriz[ind_lin])):
                print(matriz[ind_lin][ind_col], end=' ')
                ind_col+=1

        print() # adiciona um '\n' no final da linha da matriz
        ind_lin+=1

================
>> 1 2
>> 3 4
```

13 Arquivos

Arquivos são identificados por um nome e também por uma extensão, que é opcional - extensões são usadas apenas para identificar o conteúdo do arquivo (a linguagem utilizada no código, por

exemplo), e também são bastante úteis para organizar os arquivos de mesmo tipo ou selecionar programas padrão para abrir/executar arquivos. Nesse curso, falaremos apenas sobre arquivos de texto, que utilizaremos tanto para criar o código, quanto para abrir informações a serem utilizadas pelo mesmo.

Um diretório (também chamado de pasta) é um recurso organizacional do sistema, e pode conter tanto arquivos quanto outros diretórios. Essa organização nos leva à definição de **caminho absoluto** e **caminho relativo**. O primeiro é a descrição do caminho para um arquivo/diretório desde a pasta raiz do sistema, e a segunda a descrição à partir do diretório atual. Exemplos são mostrados abaixo.

```
/home/usuario/Downloads/foto.png
/curso/minicurso.pdf
```

13.1 Abrindo o arquivo

Para se trabalhar com arquivos em Python, devemos abri-lo e então associa-lo a uma variável. O primeiro passo então é abrir o arquivo com a função *open()*, indicando o nome do arquivo e o modo no qual queremos abri-lo, como abaixo. Os modos possíveis estão indicados na Tabela 1.

```
arq = open("nome_do_arquivo", 'modo')
```

Tabela 1: Modos possíveis para se abrir arquivos em Python.

Modo	**Indicador**
leitura	'r'
escrita	'w'
leitura e escrita	'r+'
append	'a'

Vale ressaltar que no exemplo acima, o 'nome_do_arquivo' pode ser indicado tanto de maneira absoluta quanto relativa; entretanto, caso seja escolhida a mandeira relativa, é necessário que o arquivo do código se encontre no diretório à partir do qual o caminho do arquivo foi descrito. Além disso, caso tentemos abrir um arquivo para leitura e ele não existir, o código retornará um erro. Se esse arquivo for aberto para escrita, um novo arquivo com o nome indicado será criado. Vale ressaltar que abrir um arquivo já existente para escrita faz com que todo o conteúdo desse arquivo seja apagado - se quisermos manter as informações, devemos utilizar o modo *append*.

13.2 Lendo o arquivo

Para ler dados de um arquivo aberto, utilizamos o método *read()*, ou o método *readline()*, ou ainda o método *readlines()*. A diferença é que o primeiro método lê um determinado número de caracteres, que deve ser passado como parâmetro - caso nenhum parâmetro seja passado, o arquivo inteiro é lido. Já o segundo lê todos os caracteres até a ocorrência do caractere '\n'. O último lê o arquivo inteiro, separando os caracteres nas ocorrências do caractere '\n'.

Quando o arquivo é aberto, um **indicador de posição** é automaticamente criado. Esse indicador é incrementado a cada chamada do método *read()*, e o valor incrementado é a quantidade de caracteres que foi lida. Caso o indicador de posição já esteja no final do arquivo, o método retorna uma *string* vazia.

Além disso, o método *close()* deve ser usado sempre que o arquivo deixar de ser usado. Isso garante que todas as possíveis alterações feitas no arquivo sejam efetivamente salvas, e também libera os recursos utilizados para a alocação do conteúdo do arquivo. Abaixo, um exemplo de código que lê todo o conteúdo do arquivo **"teste.txt"**.

```
arq = open("teste.txt", 'r')

linhas = arq.readlines()

arq.close()
```

13.3 Escrevendo no arquivo

Para escrever em um arquivo, utilizamos o método *write()*. Lembrando, para escrever em um arquivo, precisamos abri-lo no modo apropriado: 'w' para começar um novo arquivo (ou sobrescrever um já existente); 'a' para adicionar informaçõs no final do arquivo; e 'r+' para alterar os dados de um arquivo já existente. Abaixo, um exemplo para cada um dos casos.

```
arq = open("teste.txt", 'w')
arq.write("Apaguei o arquivo e escrevi um novo.\n")
arq.close()

arq = open("teste.txt", 'a')
arq.write("Adicionei uma nova linha nesse arquivo.\n")
arq.close()

arq = open("teste.txt", 'r+')
arq.write("Alterei o começo do arquivo.\n")
arq.close()
```

O código acima resulta no seguinte arquivo.

```
Alterei o começo do arquivo.
 novo.
Adicionei uma nova linha nesse arquivo.
```

14 Tratamento de erros e exceções

Algumas vezes na execução do programa, é possível que ocorram erros, como quando tentamos acessar uma posição inválida em uma lista, ou tentamos usar uma variável que ainda não tinha sido declarada, por exemplo. Até agora, dizíamos que o código retornava um erro e encerrava. À partir de agora, vamos aprender a manipular esses erros - em Python, chamamos esses erros de **exceções** (ou *exceptions*, em inglês).

Para poder manipular esses erros, utilizamos um recurso em Python chamado de *try/except*, como na estrutura abaixo. No bloco de comandos do *try*, colocamos o código que queremos executar. No bloco de comandos do *except*, colocamos o que queremos que aconteça caso ocorra algum erro. É possível adicionar diversos *excepts* para um mesmo *try*, para diferentes tipos de erro. O exemplo abaixo mostra isso.

```
lista = [10, 20, 30, 40, 50]

continua = True

while(continua):
  try:
    print("A lista tem 5 elementos, qual voce deseja?")
    ind = int(input()) - 1
    print("Seu valor eh {:.2f}! Qual seu divisor?".format(lista[ind]))
    div = int(input())
    print("{:.2f}/{:.2f}={:.2f}".format(lista[ind],div,lista[ind]/div))
    print("Deseja continuar? [S/N]")
    resp = input().upper()
    if(resp == 'S'):
      continua = True
    else:
      continua = False
  except IndexError:
    print("Índice inválido! Tente novamente.")
  except ZeroDivisionError:
    print("Divisor 0 não é valido! Tente novamente.")
```

As principais exceções em Python estão descritas a seguir.

1. *ImportError*: quando se tenta importar um módulo não existente;
2. *IndexError*: quando o índice é inválido;
3. *NameError*: quando a variável não é encontrada;
4. *SyntaxError*: quando acontece um erro de sintaxe;
5. *IndentationError*: quando o código não está corretamente identado;
6. *TabError*: quando ocorre uso inconsistente de 'tabs' e 'espaços' na identação;
7. *TypeError*: quando o tipo da variável não suporta uma determinada operação;
8. *ValueError*: quando uma função ou método recebe um parâmetro inapropriado;
9. *ZeroDivisionError*: quando se tenta dividir por 0.

15 Bibliotecas Importantes

1. Biblioteca *math*: Documentação;
2. Biblioteca *time*: Documentação;
3. Biblioteca *os*: Documentação;
4. Biblioteca *sys*: Documentação;
5. Biblioteca *numpy*: Documentação;
6. Biblioteca *matplotlib*: Documentação;
7. Biblioteca *csv*: Documentação.