Minicurso Python

Elias Luiz da Silva Júnior

CEFET-MG

O que é Python

- Python é uma linguagem interpretada de propósito geral de alto nível.
- Seu foco é na concisão e legibilidade do código.

O que é Python


```
1 print "Hello World"
```

Python 2

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

Python 3

O que é Python

- Multiparadigma:
  - Estruturada;
  - Orientada a objetos;
  - Funcional.
- Dinamicamente tipada;
- Fortemente tipada;
- Código livre.

O que é Python

Usado em diversas aplicações diferentes, como:

- Sites: Google, Youtube, Pinterest, DropBox, Reddit, Instagram, Spotify;
- Criação de scripts: Linux, Blender, DICE (Battlefield 2), 2kGames (Civilization IV);
- Computação científica: NASA, Los Alamos National Laboratory, National Weather Service.

O que é Python

Vantagens:

- Desenvolvimento rápido;
- Grande repositório de módulos (bibliotecas);
- Menos propenso a erros que outras linguagens semelhantes; *
- Mais rápido que outras linguagens semelhantes. **

*Linguagens semelhantes: PHP, Ruby, Perl, Javascript.
**Linguagens semelhantes: PHP, Ruby, Perl, Javascript (exceto V8 engine).

O que é Python

Módulos para finalidades diversas:

- aima-python: Algoritmos de inteligência artificial;
- django: Framework de desenvolvimento Web MVC;
- stochpy: Modelagem de processos estocásticos;
- numpy: Cálculo numérico e matricial;
- pycrypto: Algoritmos de criptografia.

E outros 67958 pacotes listados no repositório PyPI.

Comparação

```c
int main()
{
    int i, vetor[10];
    for(i = 0; i < 10; i++)
    {
        vetor[i] = i * 10;
    }
}
```

C

Comparação


```java
public class Main{
    public static void main(String[] args){
        int[] vetor = new int[10];
        for(int i = 0; i < 10; i++){
            vetor[i] = i * 10;
        }
    }
}
```

Java

Comparação


```php
<?php
    $vetor = range(0, 90, 10);
?>
```

PHP

```python
vetor = range(0, 100, 10);
```

Python

Comparação

```php
if ($idade >= 18 && $idade < 70) {
    echo "Voto obrigatorio"
} elseif ($idade >= 16) {
    echo "Voto facultativo"
} else {
    echo "Nao pode votar"
}
```

PHP

```python
if idade >= 18 and idade < 70:
    print("Voto obrigatorio")
elif idade >= 16:
    print("Voto facultativo")
else:
    print("Nao pode votar")
```

Python

Comparação


```php
<?php
$a = "String";
for ($i = 1; $i <= strlen($a); $i++){
    echo substr($a, 0, $i) . "<br>";
}
?>
```

PHP

```python
a = "String"
for i in range( len(a) + 1):
    print( a[:i] )
```

Python

Identação

O design Python tem como objetivo tornar a linguagem o mais legível possível. Uma das maneiras de fazer isso é forçando a identação do código.

Para isso, Python usa a identação do código para determinar os blocos de execução.

Identação


```
1  if False:
2      print("Dentro do if")
3      print("Dentro do if")
4
5  if False:
6      print("Dentro do if")
7  print("Fora do if")
8
9  if False:
10 print("Fora do if")
11 #Erro de compilação: bloco vazio
```

Identação delimitando blocos

Identação

É importante manter a consistência no código.

Por padrão são usados 4 espaços para marcar os blocos.

Tabs podem ser usados desde que não misturados com espaços. Se utilizado, o ideal é configurar o editor para substituir tabs por espaços.

Variáveis e Tipos

Python é uma linguagem dinamicamente tipada.
Isso quer dizer que o tipo da variável é determinado pelo valor a ela atribuído e pode mudar em tempo de execução.

Variáveis e Tipos


```
1 a = 1 #tipo int
2
3 a = 3.2 #tipo float
4
5 a = "abc" #tipo string
6
7 a = [] #tipo list
```

Tipagem dinâmica

Variáveis e Tipos

Python é fortemente tipado.
Isso quer dizer que o tipo da variável determina quais operações ela pode realizar e com quais tipos.
Para converter entre tipos (exceto *int* e *float*) é necessário explicitar o *casting*. Esse decisão de design foi tomada para reduzir erros e evitar certos comportamentos inesperados.

Variáveis e Tipos


```
1  a = 1 #int
2
3  a = a + 2 #permaneçe int
4
5  a = a + 2.1 #casting para float
6
7  a = a + "a" #casting implícito para string
8  #TypeError
9
10 a = str(a) + "a" #casting explícito para string
```

Tipagem forte

Variáveis e Tipos


```
1  var vetor = [];
2
3  alert(vetor == false); //verdadeiro
4
5  if (vetor)
6  alert('entrou no if'); //executa alert
7
8  '' == '0' //falso
9  0 == ''   //verdadeiro
10 0 == '0'  //verdadeiro
11
12 '\n\r\t' == 0 //verdadeiro
```

Tipagem fraca em Javascript

Estruturas condicionais

As estruturas condicionais em Python se comportam como na maioria das outras linguagens

- Não há necessidade de parentêses na condição;
- Operadores lógicos escritos em vez de símbolos:
  - “and” equivale a “&&”;
  - “or” equivale a “||”;
  - “not” equivale a “!”;
- Comparadores são iguais: $<, <=, >, >=, ==, != .$

Estruturas condicionais


```
1  idade = 21
2
3  if idade >= 18:
4      if idade >= 70:
5          print("Voto facultativo")
6      else:
7          print("Voto obrigatorio")
8  elif not idade < 16:
9      print("Voto facultativo")
10 else:
11     print("Nao pode votar")
```

Estrutura condicional

Laço while

O laço de repetição “enquanto” em Python também se comporta como na maioria das outras linguagens.

Laço while


```
1 contador = 0
2
3 while contador < 10:
4     contador += 1
5     print(contador)
```

Estrutura enquanto

Laço for

O laço de repetição “para” em Python se comporta como o laço *foreach*.

O operador *in* é utilizado para selecionar cada elemento em um iterável. Também pode ser usado para saber se um elemento está dentro de uma estrutura.

Para se iterar por numéros usamos a função *range*, que retorna um iterável contendo uma sequência de números.

Laço for

```
1 for i in range(0,10):
2     print(i)
```

em Python é equivalente a

```
1 for(int i = 0; i < 10; i++){
2     System.out.println(i);
3 }
```

em Java.

Laço for


```python
#explicitando o incremento para a funcao range
for i in range(0,10,2):
    print(i)
```

em Python é equivalente a

```java
for(int i = 0; i < 10; i+=2){
    System.out.println(i);
}
```

em Java.

Laço for

Essa mesma maneira de usar se aplica para qualquer estrutura que seja iterável:

- Strings;
- Tuplas;
- Listas;
- Dicionários.

E com algumas adaptações objetos também.

Laço for


```
1 lista = [1,2,3,4,5]
2
3 for i in lista:
4     print(i)
```

Laço for

Laço for

Os elementos armazenados na variável pelo comando *in* não alteram a posição do iterável.

Para isso pode se usar a função *range* em conjunto com a função *len*, que retorna o tamanho da estrutura.

Pode se também utilizar a função *enumerate*, que retorna o índice da iteração e o elemento.

Laço for


```
1 lista = [1,2,3,4,5]
2
3 for i in range(len(lista)):
4     lista[i] += 1
5
6 for indice, elemento in enumerate(lista):
7     print("Posicao", indice, "=", elemento)
```

Laço for

Laço for

Para funções como *enumerate* que retornam mais de um valor pode se usar `_` (underline) para não associar esse valor a nenhuma variável.

Isso é uma boa prática para quando um ou mais dos valores retornados não será utilizado, evitando criar variáveis inúteis.

Laço for

```
1 lista = [1,2,3,4,5]
2
3 for i in range(len(lista)):
4     print(lista[i])
```

é equivalente a

```
1 lista = [1,2,3,4,5]
2
3 for i, _ in enumerate(lista):
4     print(lista[i])
```

Blocos try e except

Muitas funções lançam excessões quando não se comportam da maneira esperada. Quando se executa funções desse tipo é recomendado chamá-las dentro de um bloco *try*.

Para detectar excessões lançadas com o comando *raise* colocamos logo após o bloco *try* um bloco *except*, que será executado assim que o erro que ele especifica for lançado no bloco *try*.
Caso deseje que algum código execute sendo ou não sendo lançada uma excessão esse código deve ficar dentro de um bloco *finally*.

Blocos try e except


```
1  try:
2      print(10 / 0)
3  except ZeroDivisionError:
4      print("ops...")
5
6  try:
7      raise Exception('Deu errado')
8  except Exception as e:
9      print(e)
10 finally:
11     print("Exemplo de bloco try-except")
```

Blocos try e except

Strings

Strings são objetos que armazenam texto.

Se comportam como uma sequência de caracteres que podem ser percorridos por um laço.

Por ser um objeto possui alguns métodos para facilitar sua manipulação.

Strings

Alguns métodos de string:

- string.lower(): retorna a string em letras minúsculas;
- string.upper(): retorna a string em letras maiúsculas;
- string.find(s): retorna a primeira aparição de s na string;
- string.split(s): retorna uma lista de strings resiltante da divisão da string em cada ocorrência de s;
- string.replace(s, x): substitui as ocorrências de s em string por x;

Strings


```
1  a = "Minha String"
2
3  print(a.lower())
4
5  print(a.upper())
6
7  print(a.find('String'))
8
9  print(a.split(' '))
10
11 print(a.replace('Minha', 'Outra'))
12
13 print(a[:5])
```

Strings

Strings

Strings podem ser geradas através do método *format*.

Usa se uma string como “molde”, colocando {} no lugar que deseja substituir o respectivo parâmetro.

Também podem ser usados números para indicar qual o número do parâmetro a ser substituído naquela posição.

Strings


```
1 molde = "{} eh uma linguagem {}"
2 print(molde.format('Python', 'facil'))
3
4 molde = "{0} eh uma linguagem {1}mente {1}"
5 print(molde.format('Python', 'facil'))
6
7 molde = "{a[0]} eh uma linguagem {a[1]}mente {a[1]}"
8 print(molde.format(a = ('Python', 'facil')))
```

Método format de String

Tuplas

Tuplas são uma estrutura de dados que se comporta como um vetor. A diferença é que elas são imutáveis.

Normalmente são usadas para “empacotar” dados com alguma relação ou para garantir que os dados não serão alterados.

Tuplas

```
1 a = ('um', 1)
2 b = ('dois', 2)
3
4 c = (a, b)
5
6 print(c)
7
8 for i, j in c:
9     print(i, '=', js)
```

Tuplas

Listas

Listas são uma estrutura de dados que armazenam uma sequência de valores. Diferentemente das tuplas, listas podem ser alteradas.

Substitui boa parte das estruturas de outras linguagens, podendo ser usada como vetor, lista, fila e pilha.

Listas


```
1 a = [0,1,2,3]
2
3 #adicionando um elemento
4 a.append(4)
5
6 #concatenando com outra lista
7 a = a + [5, 6]
8
9 print(a)
```

Listas

Listas

Para se selecionar posições de uma lista, tupla ou string são utilizados colchetes.

Para selecionar mais de uma posição, são passados parâmetros separados por “:”. O primeiro parâmetro determina a primeira posição a ser retornada, o segundo parâmetro determina qual posição servirá como parada e o terceiro qual passo para escolher as posições.

Os parâmetros podem ser omitidos, assumindo então seus valores padrão. Para o primeiro é a primeira posição da lista, para o segundo é a última e para o terceiro é 1.

Listas


```
1  a = list(range(0,10))
2
3  #exibindo da posicao 2 a 5
4  print(a[2:6])
5
6  #exibindo os primeiros 5 elementos
7  print(a[:5])
8
9  #exibindo da posicao 5 para frente
10 print(a[5:])
11
12 #exibindo as posicoes pares
13 print(a[::2])
14
15 #exibindo de trás para frente
16 print(a[::-1])
```

Sublistas

Listas

Alguns métodos de lista:

- lista.append(x): insere x ao fim da lista;
- lista.insert(i, x): insere x na posição i;
- lista.remove(x): remove o elemento x da lista;
- lista.pop(): remove o último elemento da lista;
- lista.pop(i): remove elemento na posição i;
- lista.index(x): retorna a posição do elemento x;

Dicionários

Dicionários são uma das estruturas de dados básicas de Python, juntamente às tuplas e listas.

Dicionários são tabelas hash dinâmicas, ou seja, são como listas mas os elementos não são identificados por sua posição, mas sim por uma “chave”. Essa chave pode ser um valor de qualquer tipo básico.

Dicionários


```
1  #criando um dicionario
2  constantes = {
3      'pi' : 3.14159,
4      'e' : 2.71828
5  }
6
7  #exibindo os valores salvos
8  print(constantes['pi'])
9  print(constantes['e'])
10
11 #exibindo o dicionario completo
12 print(constantes)
13
14 #adicionando um valor
15 constantes['phi'] = 1.61803
16 print(constantes['phi'])
```

Dicionários

Dicionários

Dicionários são iteráveis, o que quer dizer que podem ser percorridos com um laço *for*.

A função equivalente a *enumerate* para dicionários é a função *items*, que é um método do objeto dicionário.

Dicionários


```
1  constantes = {
2      'pi' : 3.14159,
3      'e' : 2.71828,
4      'phi' : 1.61803
5  }
6
7  print('Constantes disponíveis: ')
8  for chave in constantes:
9      print(chave, end = ' ')
10
11 print('\n\nValores das constantes:')
12 for chave, valor in constantes.items():
13     print(chave, "=", valor)
```

Dicionários

Funções e parâmetros

Funções em Python são criadas com a palavra-chave *def*.

Por ser uma linguagem dinamicamente tipada, as funções em Python não precisam especificar o tipo de retorno.

Funções e parâmetros


```
1 def HelloWorld():
2     print("Hello World!")
3
4 HelloWorld()
5
6 def AoQuadrado(a):
7     return a ** 2
8
9 print(AoQuadrado(2))
```

Funções

Funções e parâmetros

Python não permite a sobrecarga de funções.

Porém ele permite a definição de um valor padrão para um parâmetro e a passagem de parâmetros fora de ordem.

Funções e parâmetros


```
1 def Range(inicio = 0, fim = 10, passo = 1):
2     return list(range(inicio, fim, passo))
3
4 print(Range(1, 5))
5 print(Range(2))
6 print(Range())
7 print(Range(passo = 2))
8 print(Range(passo = 3, fim = 20))
9 print(Range(passo = 3, fim = 20, inicio = 1))
```

Parâmetros

Funções e parâmetros

Funções podem retornar mais de um valor, desde eles estejam separados por vírgula.

O que acontece realmente é que a função retorna uma tupla com os valores e depois os atribui às respectivas variàveis.

Funções e parâmetros


```
1  def RaizQuadradaCubica(valor):
2      return valor ** (1/2), valor ** (1/3)
3
4  print(RaizQuadradaCubica(64))
5
6  a, b = RaizQuadradaCubica(64)
7  print(a, b)
8
9  _,  raiz = RaizQuadradaCubica(64)
10 print(raiz)
11
12 #seguindo a mesma ideia
13 a = 1
14 b = 2
15 a, b = b, a
16 print(a, b)
```

Retorno múltiplo

Classes

Python é uma linguagem orientada a objetos, tratando tudo como objetos assim como outras linguagens como Java, a diferença é que esse tratamento nem sempre é visível para o programador, permitindo programação estruturada e funcional.

Classes são criadas com a palavra-chave *class* e possuem como construtor o método *__init__*.

Classes

A muitos conceitos de orientação a objetos se aplicam aqui, mas há algumas notáveis diferenças.

Python não permite a criação de atributos ou métodos privados por acreditar que o programador deve ter controle total do programa. Isso pode ser contornado de forma extra-oficial em alguns interpretadores colocando 2 underlines antes do nome do membro.

Outra diferença é que todos os métodos são estáticos, recebendo como primeiro parâmetro uma referência para o objeto que fez a chamada.

Classes


```
1  class Funcionario:
2      #contador estatico de funcionarios
3      contador = 1
4
5      #construtor
6      def __init__(self, nome, salario):
7
8          #self eh o nome padrao para o objeto que
               realizou a chamada
9          #equivalente ao 'this' de outras linguagens
10         #nota: pode ter outro nome, self eh somente
               um padrao
11         self.nome = nome
12         self.salario = salario
13         self.id = Funcionario.contador
14         Funcionario.contador += 1
```

Classes parte 1

Classes


```
1       #define como o objeto deve ser exibido se
            convertido para string
2       def __str__(self):
3           return "Funcionario {}\nNome: {}\nSalario:
                R${:.2f}".format(self.id, self.nome,
                self.salario)
4
5   funcionarios = {}
6   funcionarios['Jose'] = Funcionario('Jose da Silva',
        2000)
7   funcionarios['Maria'] = Funcionario('Maria da Silva'
        , 2000)
8
9   for _, i in funcionarios.items():
10      print(i)
```

Classes parte 2

Classes

Python possui um coletor de lixo para excluir da memória os objetos que não estão sendo usados.

Para forçar a exclusão de um objeto da memória utiliza-se o comando "*del* objeto".

Para determinar o comportamento de um objeto ao ser removido da memória (ex: salvar um arquivo) o método *__del__(self)* deve ser implementado.

Classes

Herança em Python se comporta como esperado na maioria das outras linguagens.

Diferente de algumas linguagens como Java e PHP, Python permite herança múltipla, não somente de interfaces mas também de classes completas, de forma parecida com C++.

Classes


```
1  class Pai:
2      def Print(self):
3          print('classe pai')
4  class Filho(Pai):
5      def Print(self):
6          print('classe filho')
7  class Filho2(Pai):
8      #declarado mas nao implementado
9      pass
10
11 a = Pai()
12 a.Print()
13 b = Filho()
14 b.Print()
15 c = Filho2()
16 c.Print()
```

Herança

Classes

Por permitir herança multipla e duck typing* Python não define uma maneira de se trabalhar com classes abstratas e interfaces.

*Duck typing: Objetos de classes diferentes podem ocupar uma mesma estrutura de dados e receber chamadas iguais.
Ex: Se uma lista *cachorros* contém objetos das classes *Labrador* e *ViraLata*, não é necessário que ambas compartilhem uma interface *Cachorro* para que eu possa chamar o método *cachorros[i].late()*.

Módulos

Bibliotecas em Python são chamadas de módulos ou pacotes, que são basicamente classes que abstraem um ou mais arquivos que podem possuir uma ou mais classes.

Para se incluir um módulo em um programa é usada a palavra-chave *import*.

Para se incluir apenas parte do módulo, como uma classe se usa "*from* módulo *import* classe".

Módulos

```
1 from MeuModulo import Funcionario
2
3 joao = Funcionario('Joao da Silva', 2000)
4 print(joao)
```

Módulos

Entrada e Saída

Entrada e saída de dados podem ser feitas de várias formas como:

- Dispositivos de IO;
- Arquivos;
- Bancos de dados;
- Dispositivos de rede;
- ...

Para escrever no dispositivo de saída padrão (monitor) utilizamos a função *print*.
Para ler do mesmo usamos a função *input*.

Entrada e Saída


```
1 valor = input("Digite alguma coisa: ")
2 print("Voce digitou: {}".format(valor))
```

Entrada e saída padrão

Entrada e Saída

Arquivos do sistema podem ser abertos utilizando a funcao *open*.
Essa função recebe o nome do arquivo e o modo de acesso (r = leitura, w = escrita, ...).

A escrita no arquivo é feita através do método *arquivo.write*.

Para ler do arquivo usamos o metodo *arquivo.read*.

Ao fim do uso do arquivo, deve ser chamado o método *arquivo.close*.

Entrada e Saída


```
1 arquivo = open("22 - teclado.py", "r")
2
3 #executa um laco for por todas as linhas do arquivo
4 for i in arquivo.readlines():
5     print(i)
6
7 arquivo.close()
```

Entrada e saída padrão

Threads

Python possui um módulo para criacao de *threads* chamado *threading*.

Threads podem ser criadas para executar uma função em paralelo ou podem ser uma classe com um método iniciando a execução.

Para executar uma função em uma *thread*, criamos um objeto do tipo *Thread* com os parâmetros *target* recebendo a função e *args* recebendo uma tupla contendo os parâmetros da função.

Threads


```
1  from threading import *
2  import time as t
3  def f(nome, atraso):
4      for i in range(4):
5          t.sleep(atraso) #para a thread por 1 segundo
6          #imprime o tempo atual
7          print(nome + ":" + t.ctime(t.time()) + "\n",
               end = "")
8  #inicia uma nova thread que executa a funcao 'f' com
       parametros "Thread 1"
9  t1 = Thread(target=f, args=("Thread 1", 1))
10 t2 = Thread(target=f, args=("Thread 2", 2))
11 t1.start()
12 t2.start()
13 t1.join() #aguarda as threads terminarem de executar
14 t2.join()
```

Função em thread

Threads

Para executar uma classe em uma *thread*, herdamos da classe *Thread* implementando o método *run*, que será chamado quando a *thread* for iniciada.

Threads


```
1  from threading import *
2  import time as t
3
4  class MinhaThread(Thread):
5      def __init__(self, nome, atraso):
6          Thread.__init__(self) #chamando o construtor
               da super-classe
7          self.nome = nome
8          self.atraso = atraso
9          def run(self):
10         for i in range(4):
11         t.sleep(self.atraso)
12         print(self.nome, ":", t.ctime(t.time()), "\n
               ", end = "")
13
14 t1 = MinhaThread("Thread 1", 1)
15 t1.start()
16 t1.join()
```

Threads

No pacote *threading* existem ainda outras classes e funções, como as classes *Lock* e *Semaphore*, usadas para fazer o sincronismo entre *threads*.

Como o objetivo deste curso é somente apresentar conceitos gerais da linguagem não nos aprofundaremos nisso.

Expressões Regulares

Expressões regulares são ferramentas muito úteis para manipulação de texto, já que elas permitem definir um padrão para uma entrada e responder se ela segue ou não esse padrão. Também são úteis para encontrar determinados padrões em entradas grandes.

Em Python o módulo que contém as funções para trabalhar com expressões regulares se chama *re*.

Expressões Regulares

Os dois métodos principais para usar expressões regulares são *match* e *search*.

A função *match* checa se a string inteira segue o padrão determinado pela expressão. Apesar de não ser um booleano, pode ser usado em if's.

Já a função *search* procura na string por trechos que sigam o padrão dado na expressão.

Expressões Regulares


```
1  import re
2  texto = """Meu telefone
3  e 8888-8888"""
4  padrao = "[0-9]{4}-[0-9]{4}"
5  #Flag re.M permite conferir em strings multilinha
6  #Flag re.I torna a busca insensível a case (nesse
       caso nao precisa)
7  resultado = re.match(padrao, texto, re.M|re.I)
8  if(resultado):
9      print("Segue o padrao")
10 else:
11     print("Nao segue o padrao")
12 resultado = re.search(padrao, texto, re.M|re.I)
13 if(resultado):
14     print("Padrao encontrado:", resultado.group())
15 else:
16     print("Padrao nao encontrado")
```

Expressões Regulares

Expressões Regulares

Caso uma expressão regular seja usada mais de uma vez é mais eficiente compilá-la e chamar os métodos do objeto resultante.

Para compilar um padrão, chame a função *re.compile* passando como parâmetro a expressão regular. O valor de retorno é um objeto que contém as funções de teste (*match, search, ...*).

Expressões Regulares


```
1  import re
2  texto = """Meu telefone
3  e 8888-8888"""
4  padrao = re.compile("[0-9]{4}-[0-9]{4}")
5  resultado = padrao.match(texto, re.M|re.I)
6  if(resultado):
7      print("Segue o padrao")
8  else:
9      print("Nao segue o padrao")
10 resultado = padrao.search(texto, re.M|re.I)
11 if(resultado):
12     print("Padrao encontrado:", resultado.group())
13 else:
14     print("Padrao nao encontrado")
```

Expressões Regulares