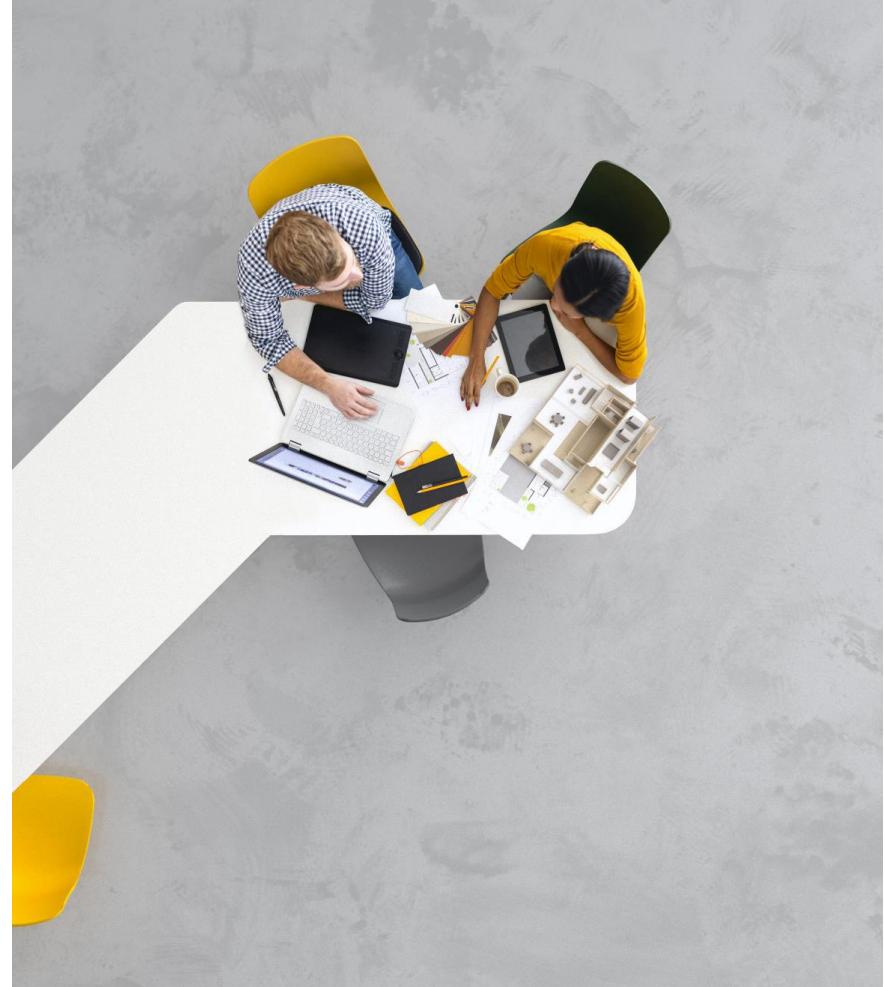


Basico de Python

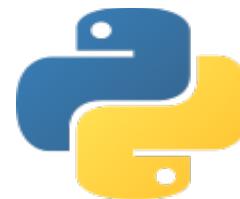
Parte II

Glauco Reis



O que é Python?

- Python é uma linguagem de programação popular de alto nível usada em várias aplicações
 - Python é uma linguagem fácil de aprender por causa de sua sintaxe simples
 - Python pode ser usado para tarefas simples, como plotagem, ou para tarefas mais complexas, como aprendizado de máquina

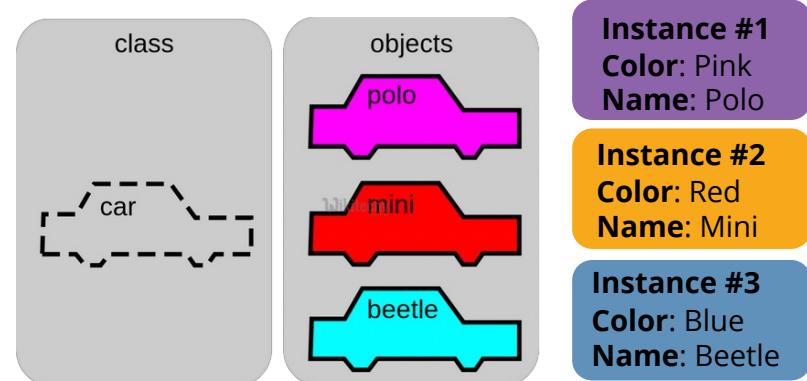


Variáveis, objetos e classes

- Uma variável é uma referência a um valor armazenado na memória de um computador.
- As variáveis podem ser classificadas em uma variedade de categorias (ou tipos de dados), como números (int/float etc), valores booleanos (verdadeiro/falso) e sequências (strings, listas etc).
- Um objeto é uma coleção de dados da memória de um computador que pode ser manipulada.
 - **TODAS AS VARIÁVEIS SÃO OBJETOS** embora alguns objetos possam ser definidos por dados referidos por múltiplas variáveis.
 - Métodos são as funções usadas para agir/alterar os dados de um objeto. Eles descrevem o que seu objeto pode "fazer".

Variáveis, objetos e classes (cont.)

- Uma classe é uma coleção de objetos que compartilham o mesmo conjunto de variáveis/métodos.
 - A definição da classe fornece um blueprint para todos os objetos dentro dela (instâncias).
 - As instâncias podem compartilhar as mesmas variáveis (cor, tamanho, forma, etc.), mas NÃO compartilham os mesmos valores para cada variável (azul/vermelho/rosa, pequeno/grande, quadrado/circular etc.)



Regras básicas de sintaxe

- O nome da sua variável (myInt etc.) é colocado à esquerda do operador "=".
 - A maioria dos nomes de variáveis está em maiúsculas e minúsculas, onde a primeira palavra começa com uma letra minúscula e todas as palavras subsequentes são maiúsculas
 - Nomes de variáveis também podem aparecer em letras maiúsculas em que todas as palavras são minúsculas, com sublinhados entre as palavras
 - O operador de atribuição ("=") define o nome da variável igual ao local de memória onde seu valor é encontrado.
- O valor da sua variável ("Hello, World") é colocado à direita do operador "=".
 - O tipo desse valor NÃO precisa ser declarado, mas seu formato deve obedecer a um determinado tipo de objeto (conforme mostrado).

```
myString = "Hello, World" myInt = 7  
myFloat = 7.0  
myList = [7, 8, 9] myBoolean = true
```

Regras básicas de sintaxe

- Sintaxe da função
 - **def**....: indica que você está definindo uma nova função.
 - **function()** refere-se ao nome da sua função. Por convenção, esse nome é tipicamente minúsculo e representa um verbo/ação.
 - **a,b** refere-se a parâmetros (valores ou variáveis) que podem ser usados dentro das instruções da definição da sua função (.....). Se sua função não tiver parâmetros, um parêntese vazio () será usado.
 - The **return** é uma instrução opcional que retornará um valor para sua função para sua chamada original.

```
def function(a, b):  
    ....  
    return a + b
```

Regras básicas de sintaxe (cont.)

- Chamando uma função
 - Chame a função referindo-se ao seu nome(**function()**) e colocando quaisquer argumentos necessários(**1, 2**) entre parênteses separados por Vírgulas. **myValue** = **function(1, 2)**
 - Se desejar, você pode definir sua chamada de função igual a uma variável(**myValue**). O valor retornado pela função será atribuído ao nome da variável.

```
myValue = function(1, 2)
```

Tipos de dados e operadores comuns

- Um tipo de dados é um meio de classificar um valor e determinar quais operações podem ser executadas nele. Todos os objetos têm um tipo de dados.
- **Operadores são símbolos usados para realizar funções/cálculos específicos.**

Operator	Description
<code>**</code>	Exponentiation (raise to the power)
<code>~ + -</code>	Complement, unary plus and minus (method names for the last two are <code>+@</code> and <code>-@</code>)
<code>* / % //</code>	Multiply, divide, modulo and floor division
<code>+ -</code>	Addition and subtraction
<code>>> <<</code>	Right and left bitwise shift
<code>&</code>	Bitwise 'AND'
<code>^ </code>	Bitwise exclusive 'OR' and regular 'OR'
<code><= < > >=</code>	Comparison operators
<code><> == !=</code>	Equality operators
<code>= %= /= //=- +=</code>	Assignment operators
<code>*= **=</code>	Identity operators
<code>is is not</code>	Membership operators
<code>in not in</code>	Logical operators
<code>not or and</code>	Logical operators

Class	Description
<code>bool</code>	Boolean value
<code>int</code>	integer (arbitrary magnitude)
<code>float</code>	floating-point number
<code>list</code>	mutable sequence of objects
<code>tuple</code>	immutable sequence of objects
<code>str</code>	character string
<code>set</code>	unordered set of distinct objects
<code>frozenset</code>	immutable form of set class
<code>dict</code>	associative mapping (aka dictionary)

Entrada/Saída

- **Funções de entrada(`input()`)** Permitir que os usuários de um programa coloquem valores no código de programação.
 - O parâmetro para uma função de entrada é chamado de prompt. Esta é uma cadeia de caracteres (isso pode ser indicado por "" ou ") como "**Enter a number:**"
 - A resposta do usuário ao prompt será retornada para a chamada de instrução de entrada como uma cadeia de caracteres. Para usar esse valor como qualquer outro tipo de dados, ele deve ser convertido com outra função(**`int()`**).
- **Print functions (`print()`)** permitem que programas produzam cadeias de caracteres para usuários em uma determinada interface.
 - O parâmetro desta função é de qualquer tipo. Todos os tipos serão convertidos automaticamente em cadeias de caracteres.

```
xString = input("Enter a number: ")
x = int(xString)
y=x+2
print(y)
```

If-else Statements

- **If-else** instruções permitem que os programadores adaptem a função de seu código com base em uma determinada condição.
- Se uma determinada condição (i.e. `x % 2 == 0`) é true, então as afirmações seguintes à instrução if (`if`) será executado. Se a condição for false, as instruções seguintes à instrução else (`else`) será executado.
 - A condição é testada usando os operadores booleanos `==` (is equal to), `!=` (is not equal to), **and** (usado para testar várias condições), and **or** (usado para testar se **AT LEAST ONE** condição é verdadeira).
 - Adicionalmente, **else-if statements** (`elif`) pode ser usado para fornecer instruções de codificação exclusivas para várias condições.

```
xString = input("Enter a number: ")
x = int(xString)
if x % 2 == 0:
    print("This is an even number")
elif x == 0:
    print("This number equals 0")
else:
    print("This is an odd number")
```

For Loops

- **For loops** executar a mesma tarefa (iterar) para o número de vezes especificado por um iterável (algo que pode ser avaliado repetidamente, como uma lista, cadeia de caracteres ou intervalo).
- **for** define o loop for
- **x** é a variável que define o número de vezes que as instruções dentro do loop (**print(myInt)**) são executados.
- O **range(start, stop, step)** função é frequentemente usada para definir x.
 - O valor inicial é definido por **start**, O valor final é definido por **stop - 1**, e a magnitude na qual x muda entre loops é definida por **step**.
- **in** é um operador booleano que retorna true se o valor dado (x) for encontrado dentro de uma determinada lista, string, range etc.

```
myString = input("Enter a number: ")  
myInt = int(myString)
```

```
for x in range(0, 5, 1): print(myInt)
```

While Loops

- **While loops** são instruções que iteram desde que uma determinada condição booleana seja atendida.
 - **x** (a variável que determina se a condição é atendida ou não) é definida e manipulada FORA do cabeçalho do loop while (**while**)
 - A condição(**x < 5**) é uma instrução que contém uma variável booleana.
 - **break** é uma instrução usada para sair do loop for/while atual.
 - **continue** é uma instrução usada para rejeitar todas as instruções na iteração de loop for/while atual e retornar ao início do loop.

```
myString = input("Enter a number: ")
myInt = int(myString)
x = 0
while x < 5:
    print(myInt)
    x= x +1
```

```
1 # Prints out 0,1,2,3,4
2
3 count = 0
4 while True:
5     print(count)
6     count += 1
7     if count >= 5:
8         break
9
10 # Prints out only odd numbers - 1,3,5,7,9
11 for x in range(10):
12     # Check if x is even
13     if x % 2 == 0:
14         continue
15     print(x)
```

Um exemplo de código (em IDLE)

```
x = 34 - 23           # A comment.  
y = "Hello"          # Another one.  
z = 3.45  
if z == 3.45 or y == "Hello":  
    x = x + 1  
    y = y + " World"  # String concat.  
print x  
print y
```

O suficiente para entender o código

- O recuo é importante para o significado do Código
- Estrutura do bloco indicada por recuo
- A primeira atribuição a uma variável cria-a
 - Os tipos de variáveis não precisam ser declarados.
 - Python descobre os tipos de variáveis por conta própria.
- Atribuição é `=` e comparação é `==`
- Para números `+ - * / %` estão como esperado
- Uso especial de `+` para concatenação de cadeia de caracteres e `%` para formatação de cadeia de caracteres (como no printf de C)
- Operadores lógicos são palavras (`and, or, not`)
não símbolos
- O comando básico de impressão é `print`

Tipos de dados básicos

- Inteiros (padrão para números)

```
z = 5 / 2 # Answer 2, integer division
```

- Floats

```
x = 3.456
```

- Strings

- Pode usar “” ou “” para especificar com “abc” == ‘abc’
 - Incomparável pode ocorrer dentro da cadeia de caracteres: “matt’s”
 - Use aspas duplas triplas para cadeias de caracteres ou cadeias de caracteres de várias linhas que contenham ' e " dentro delas:
“““a ‘b“c””””

Whitespace

Whitespace é significativo em Python: especialmente recuo e colocação de novas linhas

- Usar uma nova linha para encerrar uma linha de Código

 Use \ quando deve ir para a próxima linha prematuramente

- Sem chaves {} Para marcar blocos de código, use recuo consistente

- A primeira linha com menos recuo está fora do bloco
- A primeira linha com mais recuo inicia um bloco aninhado
- **Dois pontos iniciam um novo bloco em muitas construções, por exemplo, definições de função e, em seguida, cláusulas**

Comentários

- Iniciar comentários com #, o restante da linha é ignorado
- Pode incluir uma "cadeia de caracteres de documentação" como a primeira linha de uma nova função ou classe definida
- Ambientes de desenvolvimento, depurador e outras ferramentas usam: é bom incluir um

```
def fact(n) :  
  
    """fact(n) assume que n é um inteiro positivo e retorna  
    fatorial de n."""  
    assert(n>0)  
  
    return 1 if n==1 else n*fact(n-1)
```

Assignment

- *Vincular uma variável em Python significa definir um nome para manter uma referência a algum objeto*
- *A atribuição cria referências, não cópias*
- Nomes em Python não têm um tipo intrínseco, objetos têm tipos
- Python determina o tipo da referência automaticamente com base em quais dados são atribuídos a ela
- Criar um nome na primeira vez que ele aparece no lado esquerdo de uma expressão de atribuição:
`x = 3`
- Uma referência é excluída por meio da coleta de lixo depois que todos os nomes vinculados a ela saíram do escopo
- Python usa semântica de referência (mais adiante)

Regras de nomenclatura

- Os nomes diferenciam maiúsculas de minúsculas e não podem começar com um número. Eles podem conter letras, números e sublinhados.

bob Bob _bob _2_bob_ bob_2 BoB

- Há algumas palavras reservadas:

and, assert, break, class, continue, def, del, elif,
else, except, exec, finally, for, from, global, if,
import, in, is, lambda, not, or, pass, print, raise,
return, try, while

Convenções de nomenclatura

A comunidade Python tem estas convenções de nomenclatura recomendadas

- Minusculas para funções, métodos e atributos
- Maiusculas para constantes
- Primeira letra Maiuscula para as aulas
- “CamelCase” apenas para estar em conformidade com convenções pré-existentes
- Attributes: interface, _internal, __private

Assignment

- Você pode atribuir vários nomes ao mesmo tempo

```
>>> x, y = 2, 3
```

```
>>> x
```

2

```
>>> y
```

3

Isso facilita a trocar valores

```
>>> x, y = y, x
```

- As atribuições podem ser encadeadas

```
>>> a = b = x = 2
```

Acessando nome inexistente

Acessar um nome antes que ele tenha sido criado corretamente (colocando-o no lado esquerdo de uma atribuição) gera um erro

```
>>> y
```

```
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    y
NameError: name 'y' is not defined
```

```
>>> y = 3
```

```
>>> y
```

```
3
```

Tipos de sequência

1. Tuple: ('john', 32, [CMSC])
 - Uma simples sequência ordenada imutável de itens
 - Os itens podem ser de tipos mistos, incluindo tipos de coleção
2. Strings: "John Smith"
 - *Immutable*
 - Conceptually very much like a tuple
3. List: [1, 2, 'john', ('up', 'down')]
 - *Sequência ordenada mutável de itens de tipos mistos*

Sintaxe semelhante

- Todos os três tipos de sequência (tuplas, cadeias de caracteres e listas) compartilham muito da mesma sintaxe e funcionalidade.
- Diferença fundamental:
 - Tuples e Strings são imutáveis
 - As listas são mutáveis
- As operações mostradas nesta seção podem ser aplicadas a todos os tipos de sequência
 - a maioria dos exemplos mostrará apenas a operação executada em um

Tipos de sequência 1

- Definir tuplas usando parênteses e vírgulas

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Definir listas usando colchetes e vírgulas

```
>>> li = ["abc", 34, 4.34, 23]
```

- Defina cadeias de caracteres usando aspas (" , ' , ou """).

```
>>> st = "Hello World"
```

```
>>> st = 'Hello World'
```

```
>>> st = """This is a multi-line
```

```
string that uses triple quotes."""
```

Tipos de sequência 2

- Acessar membros individuais de uma tupla, lista ou cadeia de caracteres usando a notação "matriz" entre colchetes
- *Note que todos são baseados em 0...*

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Segundo item da tupla.
'abc'

>>> li = ["abc", 34, 4.34, 23]
>>> li[1]      # Segundo item da lista.
34

>>> st = "Hello World"
>>> st[1]    # Segundo caractere na cadeia de
             caracteres.
'e'
```

Índices positivos e negativos

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Índice positivo: contagem a partir da esquerda, começando com 0

```
>>> t[1]
```

'abc'

Índice negativo: contagem a partir da direita, começando com -1

```
>>> t[-3]
```

4.56

Slicing: Retornar cópia de um subconjunto

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Retornar uma cópia do contêiner com um subconjunto dos membros originais. Comece a copiar no primeiro índice e pare de copiar antes do segundo.

```
>>> t[1:4]
```

```
('abc', 4.56, (2,3))
```

Índices negativos contam a partir do final

```
>>> t[1:-1]
```

```
('abc', 4.56, (2,3))
```

Slicing: Retornar cópia de um =Subconjunto

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omitir o primeiro índice para fazer a cópia a partir do início do contêiner

```
>>> t[:2]  
(23, 'abc')
```

Omitir o segundo índice para fazer a cópia começando no primeiro índice e indo para o fim

```
>>> t[2:]  
(4.56, (2,3), 'def')
```

Copiando toda a sequência

- [:] faz uma cópia de uma sequência inteira

```
>>> t[:]
```

```
(23, 'abc', 4.56, (2,3), 'def')
```

- Observe a diferença entre essas duas linhas para sequências mutáveis

```
>>> l2 = l1 # Both refer to 1 ref,  
# a mudança de um afeta ambos
```

```
>>> l2 = l1[:] # Cópias independentes, duas refs
```

O Operador 'in'

- Teste booleano se um valor está dentro de um contêiner:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- Para cadeias de caracteres, testes para substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Tenha cuidado: a palavra-chave in também é usada na sintaxe de loops e comprehensões de lista

O Operador +

O operador + produz uma nova tupla, lista ou cadeia de caracteres cujo valor é a concatenação de seus argumentos.

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
```

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

```
>>> "Hello" + " " + "World"
'Hello World'
```

O Operador *

- O operador * produz uma nova tupla, lista ou cadeia de caracteres que "repete" o conteúdo original.

```
>>> (1, 2, 3) * 3  
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

```
>>> [1, 2, 3] * 3  
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> "Hello" * 3  
'HelloHelloHello'
```

As listas são mutáveis

```
>>> li = ['abc', 23, 4.34, 23]  
>>> li[1] = 45  
>>> li  
['abc', 45, 4.34, 23]
```

- Podemos alterar as listas no lugar.
- O nome li ainda aponta para a mesma referência de memória quando terminamos.

Tuples são imutáveis

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')  
>>> t[2] = 3.14
```

```
Traceback (most recent call last):  
  File "<pyshell#75>", line 1, in -toplevel-  
    tu[2] = 3.14  
TypeError: object doesn't support item assignment
```

- Você não pode mudar uma tupla.
- Você pode fazer uma nova tupla e atribuir sua referência a um nome usado anteriormente.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```
- A *imutabilidade das tuplas significa que elas são mais rápidas do que as listas.*

Operações somente em listas

```
>>> li = [1, 11, 3, 4, 5]
```

```
>>> li.append('a') # Note the method  
syntax
```

```
>>> li  
[1, 11, 3, 4, 5, 'a']
```

```
>>> li.insert(2, 'i')  
>>> li  
[1, 11, 'i', 3, 4, 5, 'a']
```

O método `extend` vs `+`

- `+` cria uma nova lista com uma nova ref de memória
- `extend` opera na lista li no lugar.

```
>>> li.extend([9, 8, 7])
```

```
>>> li
```

```
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

- *Potencialmente confuso:*

- *extend usa uma lista como argumento.*
- *append toma um singleton como argumento.*

```
>>> li.append([10, 11, 12])
```

```
>>> li
```

```
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operações somente em listas

As listas têm muitos métodos, incluindo índice, contagem, remover, reverter, classificar

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')    # index of 1st occurrence
1
>>> li.count('b')   # number of occurrences
2
>>> li.remove('b')  # remove 1st occurrence
>>> li
['a', 'c', 'b']
```

Operações somente em listas

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()      # reverse the list *in place*
>>> li
[8, 6, 2, 5]

>>> li.sort()         # sort the list *in place*
>>> li
[2, 5, 6, 8]

>>> li.sort(some_function)
# sort in place using user-defined comparison
```

Tuple details

- A vírgula é o operador de criação de tuplas, não parenteses

```
>>> 1,
```

```
(1,)
```

- Python mostra parêns para clareza (prática recomendada)

```
>>> (1,)
```

```
(1,)
```

- Não se esqueça da vírgula!

```
>>> (1)
```

```
1
```

- Vírgula à direita só é necessária para singletons outros
- Tuplas vazias têm uma forma sintática especial

```
>>> ()
```

```
()
```

```
>>> tuple()
```

Resumo: Tuples vs. Lists

- Listas mais lentas, mas mais poderosas que tuplas
 - As listas podem ser modificadas e têm muitas operações e mehtods úteis
 - Tuplas são imutáveis e têm menos recursos
- Para converter entre tuplas e listas, use as funções `list()` e `tuple()`:

```
li = list(tu)
```

```
tu = tuple(li)
```

Python na IA

Tensors

Os tensores são semelhantes aos ndarrays do NumPy, com a adição de que os tensores também podem ser usados em uma GPU para acelerar a computação.

Operações comuns para criação e manipulação desses tensores são semelhantes àquelas para ndarrays no NumPy. (rand, uns, zeros, indexação, fatiamento, remodelação, transposição, produto cruzado, produto matricial, multiplicação por elemento)

Tensors

Attributes of a tensor 't':

- t = torch.randn(1)
requires_grad - making a trainable parameter
- Por padrão: False
- Ligar:
 - o t.requires_grad_() or
 - o t = torch.randn(1,
requires_grad=True)
- Acessando o valor do tensor:
 - o t.data
- Gradiente de tensor de acesso
 - o t.grad
- grad_fn - Histórico de operações da Autograd
- t.grad_fn

```
1 import torch
2
3 N, D = 3, 4
4
5 x = torch.rand((N, D), requires_grad=True)
6 y = torch.rand((N, D), requires_grad=True)
7 z = torch.rand((N, D), requires_grad=True)
8
9 a = x * y
10 b = a + z
11 c= torch.sum(b)
12
13 c.backward()
14
15 print(c.grad_fn)
16 print(x.data)
17 print(x.grad)|
```

```
<SumBackward0 object at 0x7fd0cb970cc0>
tensor([[ 0.4118,  0.2576,  0.3470,  0.0240],
        [ 0.7797,  0.1519,  0.7513,  0.7269],
        [ 0.8572,  0.1165,  0.8596,  0.2636]])
tensor([[ 0.6855,  0.9696,  0.4295,  0.4961],
        [ 0.3849,  0.0825,  0.7400,  0.0036],
        [ 0.8104,  0.8741,  0.9729,  0.3821]])
```

Carregando dados, dispositivos e CUDA

Matrizes numpy para tensores PyTorch

- `torch.from_numpy(x_train)`
- Returns a cpu tensor! PyTorchtensor to numpy
- `t.numpy()` Using GPU acceleration
- `t.to()`
 - Sends to whatever device (cuda or cpu)
- Fallback to cpu if gpu is unavailable:
`torch.cuda.is_available()`
- Check cpu/gpu tensor OR numpyarray ?
- `type(t) or t.type()` returns
 - `numpy.ndarray`
 - `torch.Tensor`
- CPU - `torch.cpu.FloatTensor`
- GPU - `torch.cuda.FloatTensor`

Autograd

- Pacote de Diferenciação Automática
- Não precisa se preocupar com diferenciação parcial, regra de cadeia etc.
- `backward()` faz isso
- Os gradientes são acumulados para cada etapa por padrão:
 - Necessidade de zerar gradientes após cada atualização
 - `tensor.grad_zero()`

```
# Create tensors.  
x = torch.tensor(1., requires_grad=True)  
w = torch.tensor(2., requires_grad=True)  
b = torch.tensor(3., requires_grad=True)  
  
# Build a computational graph.  
y = w * x + b    # y = 2 * x + 3  
  
# Compute gradients.  
y.backward()  
  
# Print out the gradients.  
print(x.grad)    # x.grad = 2  
print(w.grad)    # w.grad = 1  
print(b.grad)    # b.grad = 1
```

Optimizer and Loss

Optimizer

- Adam, SGD etc.
- Um otimizador obtém os parâmetros
- queremos atualizar, a taxa de aprendizado que queremos usar junto com outros hiperparâmetros e realiza as atualizações

Loss

- Várias funções de perda predefinidas para escolher
- L1, MSE, Cross Entropy

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
```

```
# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([a, b], lr=lr)
```

```
for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()
```

```
    loss.backward()
```

```
    optimizer.step()
```

```
    optimizer.zero_grad()
```

```
print(a, b)
```

Model

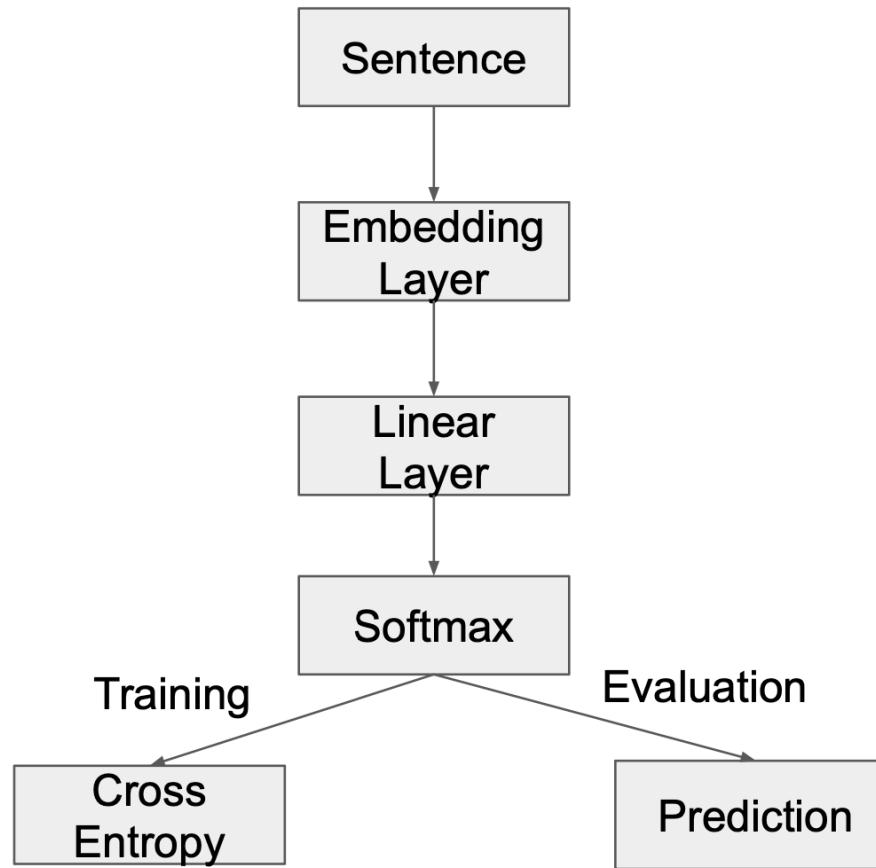
No PyTorch, um modelo é representado por uma classe Python regular que herda da classe Module.

- Dois componentes
 - `__init__(self)` : ele define as partes que compõem o modelo - no nosso caso, duas parâmetros, a e b
 - `forward(self, x)` : ele executa a computação real, ou seja, ele produz uma previsão, dado o inputx

```
class ManualLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # To make "a" and "b" real parameters of the model, we need to wrap them with nn.Parameter
        self.a = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))
        self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))

    def forward(self, x):
        # Computes the outputs / predictions
        return self.a + self.b * x
```

Overview



Design Model

- Initialize modules.
- Use linear layer here.
- Can change it to RNN,
- CNN, Transformer etc.
- Randomly initialize parameters
- Forward pass

```
import torch.nn as nn
import torch.nn.functional as F
class TextSentiment(nn.Module):
    def __init__(self, vocab_size, embed_dim, num_class):
        super().__init__()
        self.embedding = nn.EmbeddingBag(vocab_size, embed_dim, sparse=True)
        self.fc = nn.Linear(embed_dim, num_class)
        self.init_weights()

    def init_weights(self):
        initrange = 0.5
        self.embedding.weight.data.uniform_(-initrange, initrange)
        self.fc.weight.data.uniform_(-initrange, initrange)
        self.fc.bias.data.zero_()

    def forward(self, text, offsets):
        embedded = self.embedding(text, offsets)
        return self.fc(embedded)
```

Preprocess

- Criar e pré-processar conjunto de dados
- Construa vocabulário

```
import torch
import torchtext
from torchtext.datasets import text_classification
NGRAMS = 2
import os
if not os.path.isdir('./.data'):
    os.mkdir('./.data')
train_dataset, test_dataset = text_classification.DATASETS['AG_NEWS'](
    root='./.data', ngrams=NGRAMS, vocab=None)
BATCH_SIZE = 16
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
VOCAB_SIZE = len(train_dataset.get_vocab())
EMBED_DIM = 32
NUN_CLASS = len(train_dataset.get_labels())
model = TextSentiment(VOCAB_SIZE, EMBED_DIM, NUN_CLASS).to(device)
```

Preprocess

- One example of dataset:

```
print(train_dataset[0])  
  
(2, tensor([ 572, 564, 2, 2326, 49106, 150, 88, 3,  
        1143, 14, 32, 15, 32, 16, 443749, 4,  
        572, 499, 17, 10, 741769, 7, 468770, 4,  
        52, 7019, 1050, 442, 2, 14341, 673, 141447,  
        326092, 55044, 7887, 411, 9870, 628642, 43, 44,  
        144, 145, 299709, 443750, 51274, 703, 14312, 23,  
        1111134, 741770, 411508, 468771, 3779, 86384, 135944, 371666,  
        4052]))
```

- Create batch (Used in SGD)
- Choose pad or not (Using [PAD])

```
def generate_batch(batch):  
    label = torch.tensor([entry[0] for entry in batch])  
    text = [entry[1] for entry in batch]  
    offsets = [0] + [len(entry) for entry in text]  
    # torch.Tensor.cumsum returns the cumulative sum  
    # of elements in the dimension dim.  
    # torch.Tensor([1.0, 2.0, 3.0]).cumsum(dim=0)  
  
    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)  
    text = torch.cat(text)  
    return text, offsets, label
```

Training each epoch

Iterable batches

Before each optimization, make previous gradients zeros

Forward pass to compute loss

Backforward propagation to compute gradients and update parameters

After each epoch, do learning rate decay (optional)

```
from torch.utils.data import DataLoader

def train_func(sub_train_):

    # Train the model
    train_loss = 0
    train_acc = 0
    data = DataLoader(sub_train_, batch_size=BATCH_SIZE, shuffle=True,
                      collate_fn=generate_batch)
    for i, (text, offsets, cls) in enumerate(data):
        optimizer.zero_grad()
        text, offsets, cls = text.to(device), offsets.to(device), cls.to(device)
        output = model(text, offsets)
        loss = criterion(output, cls)
        train_loss += loss.item()
        loss.backward()
        optimizer.step()
        train_acc += (output.argmax(1) == cls).sum().item()

    # Adjust the learning rate
    scheduler.step()

    return train_loss / len(sub_train_), train_acc / len(sub_train_)
```

Test process

Do not need back propagation or parameter update !

```
def test(data_):
    loss = 0
    acc = 0
    data = DataLoader(data_, batch_size=BATCH_SIZE, collate_fn=generate_batch)
    for text, offsets, cls in data:
        text, offsets, cls = text.to(device), offsets.to(device), cls.to(device)
        with torch.no_grad():
            output = model(text, offsets)
            loss = criterion(output, cls)
            loss += loss.item()
            acc += (output.argmax(1) == cls).sum().item()

    return loss / len(data_), acc / len(data_)
```

The whole training process

- Use `CrossEntropyLoss()` as the criterion. The input is the output of the model. First do `logsoftmax`, then compute cross-entropy loss.
- Use SGD as optimizer.
- Use exponential decay to decrease learning rate

Print information to monitor the training process

```
import time
from torch.utils.data.dataset import random_split
N_EPOCHS = 5
min_valid_loss = float('inf')

criterion = torch.nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=4.0)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1, gamma=0.9)

train_len = int(len(train_dataset) * 0.95)
sub_train_, sub_valid_ = \
    random_split(train_dataset, [train_len, len(train_dataset) - train_len])

for epoch in range(N_EPOCHS):

    start_time = time.time()
    train_loss, train_acc = train_func(sub_train_)
    valid_loss, valid_acc = test(sub_valid_)

    secs = int(time.time() - start_time)
    mins = secs / 60
    secs = secs % 60

    print('Epoch: %d' %(epoch + 1), " | time in %d minutes, %d seconds" %(mins, secs))
    print(f'\tLoss: {train_loss:.4f}({train})\t|\tAcc: {train_acc * 100:.1f}%({train})')
    print(f'\tLoss: {valid_loss:.4f}({valid})\t|\tAcc: {valid_acc * 100:.1f}%({valid})')
```

Evaluation with testdataset or random news

```
print('Checking the results of test dataset...')
test_loss, test_acc = test(test_dataset)
print(f'\tLoss: {test_loss:.4f} (test)\t|\tAcc: {test_acc * 100:.1f}% (test)')
```

```
import re
from torchtext.data.utils import ngrams_iterator
from torchtext.data.utils import get_tokenizer

ag_news_label = {1 : "World",
                 2 : "Sports",
                 3 : "Business",
                 4 : "Sci/Tec"}

def predict(text, model, vocab, ngrams):
    tokenizer = get_tokenizer("basic_english")
    with torch.no_grad():
        text = torch.tensor([vocab[token]
                            for token in ngrams_iterator(tokenizer(text), ngrams)])
    output = model(text, torch.tensor([0]))
    return output.argmax(1).item() + 1

ex_text_str = "MEMPHIS, Tenn. - Four days ago, Jon Rahm was \
enduring the season's worst weather conditions on Sunday at The \
Open on his way to a closing 75 at Royal Portrush, which \
considering the wind and the rain was a respectable showing. \
Thursday's first round at the WGC-FedEx St. Jude Invitational \
was another story. With temperatures in the mid-80s and hardly any \
wind, the Spaniard was 13 strokes better in a flawless round. \
Thanks to his best putting performance on the PGA Tour, Rahm \
finished with an 8-under 62 for a three-stroke lead, which \
was even more impressive considering he'd never played the \
front nine at TPC Southwind."
vocab = train_dataset.get_vocab()
model = model.to("cpu")
print("This is a %s news" %ag_news_label[predict(ex_text_str, model, vocab, 2)])
```

Obrigado ! Perguntas

