Anotações da Aula de Python

Nesta aula do módulo 6, vimos...

- Conceitos Básicos de Python
- Exemplos
- Entrada, condições
- Miau
- get_positive_int
- Mario
- Transbordamento(Overflow), imprecisão
- Listas, strings
- Argumentos de linha de comando, códigos de saída
- Algoritmos
- Arquivos
- Mais bibliotecas

Quero compartilhar meu aprendizado e/ou minha dúvida...

Ir para o Discord

Recomendamos que você leia as anotações da aula, isso pode te ajudar!

Conceitos Básicos de Python

Hoje vamos aprender uma nova linguagem de programação chamada Python. Uma linguagem mais recente que C, ela possui recursos adicionais e também simplicidade, o que leva à sua popularidade.

O código-fonte em Python parece muito mais simples do que C. Na verdade, para imprimie “hello, world”, tudo o que precisamos escrever é:

```
print("hello, world")
```

- Observe que, ao contrário de C, não precisamos especificar uma nova linha na função de **print** ou usar um ponto e vírgula para terminar nossa linha.
- Para escrever e executar este programa, usaremos o IDE CS50, salvaremos um novo arquivo como hello.py apenas com a linha acima e executaremos o comando python **hello.py**.

Podemos obter strings de um usuário:

```
answer = get_string("Qual o seu nome? ")
print("ola, " + answer)
```

- Também precisamos importar a versão Python da biblioteca CS50, **cs50** , apenas para a função **get_string**, então nosso código ficará assim:

```
from cs50 import get_string

answer = get_string("Qual o seu nome? ")
print("ola, " + answer)
```

- Criamos uma variável chamada **answer** sem especificar o tipo e podemos combinar, ou concatenar, duas strings com o operador + antes de passá-lo para **print**.

Podemos usar a sintaxe para **strings de formato** , **f "..."** , para inserir variáveis. Por exemplo, poderíamos ter escrito **print(f "hello, {answer}")** para inserir o valor de **answer** em nossa string colocando-o entre chaves.

Podemos criar variáveis apenas com **counter = 0**. Ao atribuir o valor de **0**, estamos definindo implicitamente o tipo como um inteiro, portanto, não precisamos especificar o tipo. Para incrementar uma variável, podemos usar **counter = counter + 1** ou **counter + = 1**.

As condições são semelhantes a:

```
if x < y:
   print("X é menor que Y")
elif x > y:
   print("X é maior que Y")
else:
   print("X é igual a Y")
```

- Ao contrário de C, onde as chaves são usadas para indicar blocos de código, o recuo exato de cada linha é o que determina o nível de aninhamento em Python.
- E em vez de **else if**, apenas dizemos **elif**.

As expressões booleanas também são ligeiramente diferentes:

```
while True:
   print("hello, world")
```

- Ambos **True** e **False** são capitalizados em Python.

Podemos escrever um loop com uma variável:

```
i = 0
while i < 3:
    print("hello, world")
    i += 1
```

Também podemos usar um **for** loop , onde podemos fazer algo para cada valor em uma lista:

```
for i in [0, 1, 2]:
    print("cough")
```

- Listas em Python, **[0, 1, 2]**, são como matrizes em C.
- Este **for** loop irá definir a variável i para o primeiro elemento, 0 , executar, em seguida, para o segundo elemento, 1, executar, e assim por diante.
- E podemos usar uma função especial, **range**, para obter algum número de valores, como em **for i in range(3):**. **range(3)** nos dará uma lista de até, mas não incluindo 3, com os valores **0, 1** e **2**, que podemos usar. **range()** tem outras opções também, então podemos ter listas que começam com valores diferentes e têm incrementos diferentes entre os valores. Olhando a documentação , por exemplo, podemos usar **range(0, 101, 2)** para obter um intervalo de **0** a **100** (já que o segundo valor é exclusivo), incrementando em **2** de cada vez.
- Para imprimir **i** , também podemos escrever **print(i)**.
- Como geralmente há várias maneiras de escrever o mesmo código em Python, as formas mais comumente usadas e aceitas são chamadas de **Pythonic**.

Em Python, existem muitos tipos de dados integrados:

- bool , **True** ou **False**
- float , números reais
- int , inteiros
- str , strings

Enquanto C é uma **linguagem fortemente tipada**, onde precisamos especificar tipos, Python é **fracamente tipada**, onde o tipo está implícito nos valores.

Outros tipos em Python incluem:

- **range**, sequência de números
- **list**, sequência de valores mutáveis ou valores que podemos alterar

    - E as listas, mesmo que sejam como arrays em C, podem aumentar e diminuir automaticamente em Python
- **tuple**, coleção de valores ordenados como coordenadas xey ou longitude e latitude
- **dict**, dicionários, coleção de pares chave / valor, como uma tabela hash
- **set**, coleção de valores únicos ou valores sem duplicatas

A biblioteca CS50 para Python inclui:

- get_float
- get_int
- get_string

E podemos importar funções uma de cada vez ou todas juntas:

```
from cs50 import get_float
from cs50 import get_int
from cs50 import get_string
------------------------------
import cs50
------------------------------
from cs50 import get_float, get_int, get_string
```

Exemplos

Como o Python inclui muitos recursos, bem como bibliotecas de código escrito por outros, podemos resolver problemas em um nível mais alto de abstração, em vez de implementar todos os detalhes nós mesmos.

Podemos desfocar uma imagem com:

```
from PIL import Image, ImageFilter

before = Image.open("bridge.bmp")
after = before.filter(ImageFilter.BoxBlur(1))
after.save("out.bmp")
```

- No Python, incluímos outras bibliotecas com **import** e aqui vamos importar os nomes de **Image** e **ImageFilter** da biblioteca PIL. (Outras pessoas escreveram esta biblioteca, entre outras, e disponibilizaram para todos nós baixarmos e usarmos.)
- **Image** é uma estrutura que não apenas possui dados, mas funções que podemos acessar com o ".", como com **Image.open**.

- Abrimos uma imagem chamada **bridge.bmp**, chamamos uma função de filtro de desfoque e salvamos em um arquivo chamado **out.bmp**.
- E nós podemos executar isso com **python blur.py** depois de salvar para um arquivo chamado **blur.py.**

Podemos implementar um dicionário com:

```
words = set()

def load(dictionary):
   file = open(dictionary, "r")
   for line in file:
      words.add(line.rstrip())
   file.close()
   return True

def check(word):
   if word.lower() in words:
      return True
   else:
      return False

def size():
   return len(words)

def unload():
   return True
```

- Primeiro, criamos um novo conjunto chamado **words**.
- Observe que não precisamos de uma função main. Nosso programa Python será executado de cima para baixo. Aqui, queremos definir uma função, então usamos def load(). load terá um parâmetro, dictionary, e seu valor de retorno está implícito. Nós abrimos o arquivo com open, e iteramos sobre as linhas no arquivo com apenas for line in file:. Em seguida, removemos a nova linha no final da linha e a adicionamos à words. Observe que line é uma string, mas tem uma função .rstrip que podemos chamar.
- Então, para check, podemos apenas perguntar if word.lower() in words. Para o tamanho , podemos usar len para contar o número de elementos em nosso conjunto e, finalmente, para unload, não precisamos fazer nada, já que o Python gerencia a memória para nós.

Acontece que, embora implementar um programa em Python seja mais simples para nós, o tempo de execução de nosso programa em Python é mais lento do que nosso programa em C, pois a linguagem tem que trabalhar mais para nós com soluções de uso geral, como para gestão de memória.

Além disso, Python também é o nome de um programa chamado **interpretador**, que lê nosso código-fonte e o traduz para um código que nossa CPU pode entender, linha por linha.

Por exemplo, se nosso pseudocódigo da semana 0 estava em espanhol e não entendíamos espanhol, teríamos que traduzi-lo lentamente, linha por linha, para o inglês antes de podermos

pesquisar um nome em uma lista telefônica:

```
Recoge guía telefónica
Abre a la mitad de guía telefónica
Ve la página
Si la persona está en la página
Llama a la persona
Si no, si la persona está antes de mitad de guía telefónica
Abre a la mitad de la mitad izquierda de la guía telefónica
Regresa a la línea 3
Si no, si la persona está después de mitad de guía telefónica
Abre a la mitad de la mitad derecha de la guía telefónica
Regresa a la línea 3
De lo contrario
Abandona
```

Portanto, dependendo de nossos objetivos, também teremos que considerar a troca de tempo humano ao escrever um programa que é mais eficiente, versus o tempo de execução do programa.

Input, condições

Podemos obter a entrada do usuário com a função de input:

```
answer = input("Qual é seu nome? ")
print(f"olá, {answer}")
```

Podemos pedir ao usuário dois inteiros e adicioná-los:

```
from cs50 import get_int

# Solicitar x ao usuário
x = get_int("x:")

# Solicitar ao usuário y
y = get_int("y:")

# Executar adição
print(x + y)
```

- Os comentários começam com # em vez de **//**.

Se chamarmos **input** nós mesmos, obtemos strings de volta para nossos valores:

```
# Solicitar x ao usuário
x = input("x:")

# Solicitar ao usuário y
y = input("y:")

# Executar adição
print(x + y)
```

Portanto, precisamos fazer o **cast**, ou converter, cada valor de **input** em um **int** antes de armazená-lo:

```
# Solicitar x ao usuário
x = int(input( "x:" ))

# Solicitar ao usuário y
y = int(input ( "y:" ))

# Executar adição
print(x + y)
```

- Mas se o usuário não digitou um número, precisaremos fazer ainda mais verificações de erro ou nosso programa travará. Portanto, geralmente queremos usar uma biblioteca comumente usada para resolver problemas como este.

Vamos dividir os valores:

```
# Solicitar x ao usuário
x = int(input( "x:" ))

# Solicitar ao usuário y
y = int(input( "y:" ))

# Executar divisão
print(x/y)
```

- Observe que obtemos valores decimais de ponto flutuante de volta, mesmo se dividirmos dois inteiros.

E podemos demonstrar as condições:

```
from cs50 import get_int

x = get_int("x:")
y = get_int("y:")

if x < y
   print("x é menor que y")
elif x > y:
   print("x é maior que y")
else:
   print("x é igual a y")
```

Podemos importar bibliotecas inteiras e usar funções dentro delas como se fossem uma estrutura:

```
import cs50

x = cs50.get_int("x:")
y = cs50.get_int("y:")
```

- Se nosso programa precisar importar duas bibliotecas diferentes, cada uma com uma função **get_int**, por exemplo, precisaríamos usar este método para funções de **namespace** , mantendo seus nomes em espaços diferentes para evitar que colidam.

Para comparar strings, podemos dizer:

```
from cs50 import get_string

s = get_string("Você concorda?”)

if s  "Y" or s  "y":
print("Concordo")
elif s  "N" or s  "n" :
print("Não concordo.")
```

- Python não tem caracteres, então verificamos **Y** e outras letras como strings. Também podemos comparar strings diretamente com ==. Finalmente, em nossas expressões booleanas, usamos **or** e **and** em vez de símbolos.
- Também podemos dizer **if s.lower() in ["y", "yes"]:** para verificar se nossa string está em uma lista, depois de convertê-la para minúsculas primeiro.

Miau

Podemos melhorar as versões do **miau** também:

```
print("miau")
print("miau")
print("miau")
```

- Não precisamos declarar uma função principal , então apenas escrevemos a mesma linha de código três vezes.

Podemos definir uma função que podemos reutilizar:

```
for i in range(3):
miau()

def miau():
print("miau")
```

Mas isso causa um erro quando tentamos executá-lo: NameError: name 'meow' is not defined. Acontece que precisamos definir nossa função antes de usá-la, então podemos mover nossa definição de miau para o topo ou definir uma função principal primeiro:

```
def main():
for i in range(3):
```

```
miau()

def miau():
print("miau")

main()
```

- Agora, quando realmente chamarmos nossa função **main**, a função **miau** já terá sido definida.

Nossas funções também podem receber inputs:

```
def main():
miau(3)
def miau(n):
for i in range(n):
print("miau")

main()
```

- Nossa função **miau** recebe um parâmetro, **n**, e o passa para **range**.

get_positive_int

Podemos definir uma função para obter um número inteiro positivo:

```
from cs50 import get_int

def main():
i = get_positive_int()
print(i)

def get_positive_int():
while True:
n = get_int("Inteiro Positivo: ")
if n > 0:
break
return n

main()
```

- Uma vez que não há um do-while loop em Python como existe em C, temos um **while** loop que vai continuar infinitamente, e usar **break** para terminar o loop, assim que **n** > **0**. Finalmente, a nossa função vai **return n**, no nosso nível de recuo original, fora do **while** loop.
- Observe que as variáveis em Python têm funções como **variável de escopo** padrão, o que significa que **n** pode ser inicializado em um loop, mas ainda pode ser acessado posteriormente na função.

Mario

Podemos imprimir uma linha de pontos de interrogação na tela:

```
for i in range(4):
   print("?", end="")
   print()
```

- Quando imprimimos cada bloco, não queremos a nova linha automática, então podemos passar um argumento nomeado , também conhecido como argumento de palavra-chave, para a função de impressão , que especifica o valor para um parâmetro específico. Até agora, vimos apenas argumentos posicionais , onde os parâmetros são definidos com base em sua posição na chamada de função.
- Aqui, dizemos end = "" para especificar que nada deve ser impresso no final de nossa string. end também é um argumento opcional , que não precisamos passar, com um valor padrão de \ n , que é o motivo pelo qual print geralmente adiciona uma nova linha para nós.
- Finalmente, depois de imprimir nossa linha com o loop, podemos chamar **print** sem nenhum outro argumento para obter uma nova linha.

Também podemos “multiplicar” uma string e imprimi-la diretamente com: **print("?" * 4)**.

Podemos implementar loops aninhados:

```
for i in range(3):
   for j in range(3):
      print("#", end="")
      print()
```

Overflow, imprecisão

Em Python, tentar causar um overflow de inteiros não funcionará:

```
i = 1
while True:
   print(i)
   i *= 2
```

- Vemos números cada vez maiores sendo impressos, já que o Python usa automaticamente cada vez mais memória para armazenar números para nós, ao contrário de C, em que os inteiros são fixados em um determinado número de bytes.

A imprecisão de ponto flutuante também ainda existe, mas pode ser evitada por bibliotecas que podem representar números decimais com quantos bits forem necessários.

Listas, strings

Podemos fazer uma lista:

```
scores = [72, 73, 33]

print("Average: " + str(sum(scores) / len(scores)))
```

- Podemos usar **sum**, uma função incorporada ao Python, para somar os valores em nossa lista e dividi-la pelo número de pontuações, usando a função **len** para obter o comprimento da lista. Em seguida, convertemos o float em uma string antes de podermos concatená-lo e imprimi-lo.
- Podemos até adicionar a expressão inteira em uma string formatada para o mesmo efeito:

```
print(f"Average: {sum(scores) / len(scores)}")
```

Podemos adicionar itens a uma lista com:

```
from cs50 import get_int

scores = []
for i in range(3):
scores.append(get_int("Score: "))
...
```

Podemos iterar sobre cada caractere em uma string:

```
from cs50 import get_string

s = get_string("Antes: ")
print("Depois: ", end="")
for c in s:
print(c.upper(), end="")
print()
```

- Python irá iterar sobre cada caractere na string para nós com apenas **for c in s**.

Para tornar uma string maiúscula, também podemos simplesmente chamar s.upper() , sem ter que iterar nós mesmos sobre cada caractere.

Argumentos de linha de comando, códigos de saída

Podemos aceitar argumentos de linha de comando com:

```
from sys import argv
```

```
if len(argv) == 2:
print(f"hello, {argv[1]}")
else:
print("hello, world")
```

- Importamos **argv** do **sys**, ou módulo do sistema, integrado ao Python.
- Como **argv** é uma lista, podemos obter o segundo item com **argv[1]**, portanto, adicionar um argumento com o comando **python argv.py David** resultará em **olá, David** impresso.
- Como em C, **argv[0]** seria o nome do nosso programa, como argv.py.

Também podemos permitir que o Python itere sobre a lista para nós:

```
from sys import argv

for arg in argv:
   print(arg)
```

Também podemos retornar códigos de saída quando nosso programa for encerrado:

```
import sys

if len(sys.argv) != 2:
   print("missing command-line argument")
   sys.exit(1)
print(f"ola, {sys.argv[1]}")
sys.exit(0)
```

- Importamos todo o módulo **sys** agora, já que estamos usando vários componentes dele. Agora podemos usar **sys.arg** e **sys.exit()** para sair de nosso programa com um código específico.

Algoritmos

Podemos implementar a pesquisa linear apenas verificando cada elemento em uma lista:

```
import sys

numbers = [4, 6, 8, 2, 7, 5, 0]

if 0 in numbers:
print("Encontrado")
sys.exit(0)

print("Nao Encontrado")
sys.exit(1)
```

- Com **if 0 in numbers:**, estamos pedindo ao Python para verificar a lista para nós.

Uma lista de strings também pode ser pesquisada com:

```
names = ["Bill", "Charlie", "Fred", "George", "Ginny", "Percy", "Ron"]

if "Ron" in names:
print("Found")
else:
print("Not found")
```

Se tivermos um dicionário, um conjunto de pares de chaves-valores, também podemos verificar se há uma chave específica e examinar o valor armazenado para ela:

```
from cs50 import get_string

people = {
"Brian": "+1-617-495-1000",
"David": "+1-949-468-2750"
}

name = get_string("Nome: ")
if name in people:
print(f"Numero: {people[name]}")
```

- Primeiro declaramos um dicionário, **people**, onde as chaves são strings de cada nome que queremos armazenar, e o valor que queremos associar a cada chave é uma string de um número de telefone correspondente.
- Então, usamos **if name in people:** para pesquisar as chaves do nosso dicionário por um **name**. Se a chave existe, então podemos obter o valor com a notação de colchetes, **people[name]**, bem como indexar em um array com C, exceto que aqui usamos uma string em vez de um inteiro.

Os dicionários, assim como os conjuntos, são normalmente implementados em Python com uma estrutura de dados como uma tabela hash, para que possamos ter uma pesquisa de tempo quase constante. Novamente, temos a desvantagem de ter menos controle sobre exatamente o que acontece nos bastidores, como poder escolher uma função hash, com a vantagem de ter que fazer menos trabalho nós mesmos.

A troca de duas variáveis também pode ser feita simplesmente atribuindo os dois valores ao mesmo tempo:

```
x = 1
y = 2

print(f"x is {x}, y is {y}")
x, y = y, x
print(f"x is {x}, y is {y}")
```

Em Python, não temos acesso a ponteiros, o que nos protege de cometer erros com a memória.

Arquivos

Vamos abrir um arquivo CSV:

```
import csv

from cs50 import get_string

file = open("phonebook.csv", "a")

name = get_string("Nome: ")
number = get_string("Numero: ")

writer = csv.writer(file)
writer.writerow([name, number])

file.close()
```

- Acontece que Python também tem uma biblioteca **csv** que nos ajuda a trabalhar com arquivos CSV, então, depois de abrir o arquivo para anexar, podemos chamar **csv.writer** para criar um **writer** do arquivo, o que oferece funcionalidade adicional, como **writer.writerow** para escrever uma lista como uma linha.

Podemos usar a palavra-chave **with**, que fechará o arquivo para nós quando terminarmos:

```
...
with open("phonebook.csv", "a") as file:
    writer = csv.writer(file)
    writer.writerow((name, number))
```

Podemos abriir outro arquivo CSV...

```
import csv

houses = {
   "Gryffindor": 0,
   "Hufflepuff": 0,
   "Ravenclaw": 0,
   "Slytherin": 0
}

with open("Sorting Hat (Responses) - Form Responses 1.csv", "r") as file:
   reader = csv.reader(file)
   next(reader)
   for row in reader:
      house = row[1]
      houses[house] += 1

for house in houses:
   print(f"{house}: {houses[house]}")
```

Usamos a função de **reader** da biblioteca **csv**, pulamos a linha do cabeçalho com **next(reader)** e, em seguida, iteramos sobre cada uma das linhas restantes.

- O segundo item em cada linha, **row[1]**, é a string de uma casa, então podemos usá-la para acessar o valor armazenado em **houses** para aquela chave e adicionar um a ela.
- Por fim, imprimiremos a contagem de cada casa.

Mais bibliotecas

Em nosso próprio Mac ou PC, podemos abrir um terminal após instalar o Python e usar outra biblioteca para converter texto em fala:

```
importar pyttsx3

engine = pyttsx3.init ()
engine.say("olá, mundo")
engine.runAndWait()
```

- Lendo a documentação, podemos descobrir como inicializar a biblioteca e dizer uma string.
- Podemos até passar uma string de formato com **engine.say (f "olá, {nome}")** para dizer alguma entrada.

Podemos usar outra biblioteca, **face_recognition**, para encontrar rostos nas imagens:

```
# Encontre rostos na imagem
# https://github.com/ageitgey/face_recognition/blob/master/examples/find_faces_in_picture.py
from PIL import Image
import face_recognition

# Carregue o arquivo jpg em uma matriz numpy
image = face_recognition.load_image_file("office.jpg")

# Encontre todos os rostos na imagem usando o modelo baseado em HOG padrão.
# Este método é bastante preciso, mas não tão preciso quanto o modelo CNN e não é acelerado por
GPU.
# Veja também: find_faces_in_picture_cnn.py
face_locations = face_recognition.face_locations(image)

for face_location in face_locations:
   # Imprima a localização de cada rosto nesta imagem: cima, direita, baixo, esquerda
   top, right, bottom, left = face_location

   # Você pode acessar o próprio rosto desta forma:
   face_image = image [top: bottom, left: right]
   pil_image = Image.fromarray(face_image)
   pil_image.show()
```

Com **recognize.py** podemos escrever um programa que encontra uma correspondência para um rosto específico.

Podemos criar um código QR, ou código de barras bidimensional, com outra biblioteca:

```
import os
import qrcode

img = qrcode.make("https://youtu.be/oHg5SJYRHA0")
img.save("qr.png","PNG")
os.system("open qr.png")
```

Podemos reconhecer a entrada de áudio de um microfone:

```
import speech_recognition

# Obtenha áudio do microfone
recognizer = speech_recognition.Recognizer()
with speech_recognition.Microphone() como fonte:
   print("Say something:")
   audio = reconhecizer.listen(fonte)

# Reconhecer fala usando o Google Speech Recognition
print("Você disse:")
print(recognizer.recognize_google(audio))
```

Estamos seguindo a documentação da biblioteca para ouvir nosso microfone e convertê-lo em texto.

Podemos até adicionar lógica adicional para respostas básicas:

```
...
words = recognizer.recognize_google(audio)

# Responda ao discurso
if "hello" in words:
   print("Olá para você também :)")
elif "how are you" em palavras:
   print("Estou bem, obrigado!")
elif "goodbye" em palavras:
   print("Tchau pra você também")
else:
   print("Hm?")
```

- Finalmente, usamos outro programa mais sofisticado para gerar deepfakes, ou vídeos que parecem realistas, mas gerados por computador, de várias personalidades.
- Tirando proveito de todas essas bibliotecas disponíveis gratuitamente online, podemos facilmente adicionar funcionalidades avançadas aos nossos próprios aplicativos.