Parei na página { 179 } do how to automate brong stuff with python.

C. 2 [p.31] - FLOW CONTROL:

LOOPS:

While:

O ‘while’ executa em quanto determinada condicao for verdade então para encerrar ele precisa criar uma condição a ser atingida para que seja falso.

nome = *None  
while* nome != 'seu nome':  
 print('por favor digite seu nome')  
 nome = input()  
print('Obrigado')

só vai deixar de ser verdade quando digitar ‘seu nome’ ai ele sai do bloco ‘while’

Levando isso em conta, o código seguinte é outra maneira de escrever a mesma ação

*while True*:  
 nome = str(input('Digite seu nome: '))  
 *if* nome == 'seu nome':  
 print('Obrigado')  
 *break*

nesse caso é criado um loop infinito e o programa checa no bloco ‘if’ se o texto digitado é o mesmo que ‘seu nome’, se for ele executa o ‘break’ que é usado para sair do loop

CASO O PROGRAMA ENTRE EM UM LOOP INFINITO O COMANDO CRTL+C É USADO PARA SAIR DELE

Aqui está outro exemplo do ‘while’, só que agora usando o continue:

*while True*:  
 nome = str(input('Quem é voce? '))  
 *if* nome != 'leo':  
 *continue* senha = str(input('Beleza, qual a senha? '))  
 *if* senha == 'senha':  
 *break*print('voce é o bichao mesmo hein doido')

O continue serve para executar de novo o bloco ao qual esta identado, ele diz “execute de novo se passar por aqui(continue)”, é como se fosse o contrario do ‘break’, que diz “saia do bloco que eu estava executando”.

VALORES FALSOS: O PYTHON CONSIDERA O ‘’ (str vazia), 0 e 0.0 COMO SENDO FALSOS

For:

O loop do tipo ‘for’ serve para executar o código identado apenas uma quantidade determinada de vezes, do contrario do ‘while’ que executa em quanto uma condição for verdade, como no exemplo a seguir:

print('Olá!')  
*for* i *in* range (1, 4):  
 print('meu nome é leo "tres vezes" - ({})'.format(i))  
 i += 1

Aqui ele executa o código 3 vezes conforme definido no método ‘range’ (esta de 1 a 4 no caso pois ele conta até o numero anterior do 4) e para aumentar em 1 o valor da variável ‘i’ cada vez que for executao foi adicionado ao bloco identado o i += 1 que diz para o código adicionar 1 ao valor da variável 1 cada vez que o código rodar.

No exemplo seguinte podemos usar o loop ‘for’ para fazer a soma gaussiana usando força bruta:

total = 0  
*for* num *in* range (101):  
 total = total + num  
print('Os numeros somados de 0 a 100 equivalem a: {}'.format(total))

A soma gaussiana é caracterizada pela soma de todos os números de 0 a 100. Gauss descobriu um truque para fazer essa soma de maneira mais simples mas ja que temos um computador a nossa disposição podemos fazer isso de maneira mais ‘forçada’.

Mexendo agora um pouco com a função ‘range()’, ela pode ser chamada usando 3 parametros (argumentos), o primeiro e o segundo sendo o valor inicial, o segundo o valor final, e o terceiro o valor que ela vai contar por vez que rodar, por exemplo:

*for* i *in* range(0,11,2):  
 print(i)

Nesse exemplo o loop vai ser executado de 0 ate 11 contando de 2 em 2, no caso ele dá ‘print’ nos valores pares, o mesmo pode ser feito para os valores impares alterando um pouco o programa, como segue:

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

Obs: valores negativos no terceiro parâmetro fazem a função fazer uma contagem regressiva.

IMPORTANDO METODOS:

Para importar um método ele deve obedecer a sintaxe como no código abaixo:

*import* random  
vezes = int(input('Quantas vezes deseja rolar o d20? : '))  
*for* i *in* range(vezes):  
 d20 = random.randint(1, 21)  
 *if* d20 == 20:  
 print('Acerto crítico! Você rolou: {} =D'.format(d20))  
 *elif* d20 == 1:  
 print(print('Erro crítico! Você rolou: {} =['.format(d20)))  
 *else*:  
 print('Você rolou: {} '.format(d20))

Nesse exemplo importamos o modulo ‘random’ da biblioteca padrão do python e usamos sua função ‘randint()’. Para usar uma função você deve escrever o método de onde ele vem ‘.’ Sua função a ser usada.

Há duas maneiras de importar uma função especifica para ser usada no código, a primeira é simplesmente usando o ‘import funcaoX’ como visto acima e o segundo é ‘from Modulo import funcaoX’:

*from* random *import* randint  
vezes = int(input('Quantas vezes deseja rolar o d20? : '))  
*for* i *in* range(vezes):  
 d20 = randint(1, 21)  
 *if* d20 == 20:  
 print('Acerto crítico! Você rolou: {} =D'.format(d20))  
 *elif* d20 == 1:  
 print(print('Erro crítico! Você rolou: {} =['.format(d20)))  
 *else*:  
 print('Você rolou: {} '.format(d20))

Dessa maneira você não precisa indicar de qual modulo está importando a função porem o código tende a ficar menos legível então sempre que possível use a primeira maneira, ou faça como desejar.

Outro modulo interessante para se importar é o ‘sys’, ele tras funções do sistema, veja:

*import* sys  
*while True*:  
 sair = str(input('digite "sair" para sair: '))  
 *if* sair == 'sair':  
 sys.exit()

Aqui o programa entra em um loop infinito e em quanto o usuário não digitar ‘sair’ o programa continua executando, o ‘exit()’ sai do programa.

C. 3 [p.61] - FUNÇÕES:

Funções são um jeito de unir um conjunto de instruções em código em um único bloco que pode ser utilizado ao longo do código; funciona da mesma maneira que funções já conhecidas como o ‘print()’.

Para criar uma função utilizamos a palavra ‘def’ e então declaramos o nome da função; o nome da função sempre é seguido pelo ‘()’ que pode ou não conter parâmetros para o funcionamento desta função, estes parâmetros nada mais são que variáveis que operam somente dentro da função, ao chamar a função (executa-la no código) os parâmetros são chamados de argumentos; então começa a indentação com as instruções de como essa função se comportará; o que vem depois de ‘return’ é o que a função dará de output. Vide código a seguir:

*import* random  
*def* bolaOito(*number*):  
 *if number* == 1:  
 *return* 'Com certeza!'  
 *elif number* == 2:  
 *return* 'Sim será.'  
 *elif number* == 3:  
 *return* 'Sim.'  
 *elif number* == 4:  
 *return* 'Está meio nublado, tente de novo.'  
 *elif number* == 5:  
 *return* 'Me pergunte depois.'  
 *elif number* == 6:  
 *return* 'Concentre-se e pergunte depois.'  
 *elif number* == 7:  
 *return* 'Talvez.'  
 *elif number* == 8:  
 *return* 'Não.'  
r = random.randint(1, 8)  
print(r)  
print(bolaOito(r))

Esse programa funciona como uma Bola8 que te dá respostas baseadas em um número predefinido, importamos e usamos o método ‘random.randint’ para gerar o número aleatório e criamos a função ’ bolaOito()’ e aplicamos esse número, delegado a uma variável chamada ‘r’, como argumenta desta função

Aqui é importante freezar que a chamada de uma função pode ser usada em uma expressão pois seu valor equivale ao valor de retorno dela.

None:

‘None’ é um tipo de valor em Python assim como Boolean ou string e ele representa a ausência de valor; por se tratar de um tipo de valor em si ele deve ser escrito com a letra inicial maiúscula, ou seja, None.

Uma interação curiosa do None é com o print. O print() é uma função que imprime na tela o valor dentro do argumento e justamente por fazer só isso ela não tem valor retornado, por isso no código a seguir o valor booleano que None == print(‘ola’) retorna é True.

spam = print ('Olá')  
*if* spam == *None*:  
 print(*True*)  
*else*:  
 print(*False*)

Argumentos:

Quando se tratando de variáveis é importante discernir dois escopos: local e global. De maneira resumida o local é dentro da indentação de funções e o global é fora dele, ou seja, dentro de todo o resto.

O local lida diretamente com os parâmetros e variáveis criados dentro de uma função, são variáveis que só agem em quanto a função é executada, sendo assim não podem sair da indentação dela e por isso não são acessíveis fora dela, o único jeito de acessar alguma coisa que esteve dentro de uma função é através do return dela.

Todos as variáveis criadas fora do escopo local, ou seja, dentro de funções, são variáveis que agem no escopo global e podem ser acessadas no programa a qualquer momento.

Uma variável só pode ser uma das duas coisas, ou é global ou é local, nunca as duas.

Sendo assim, o escopo local pode acessar variáveis de escopo global mas o global não pode acessar as de locais e diferentes escopos locais também não tem acesso entre si; cada escopo local é um universo a parte em si. Só existe um escopo global, que é iniciado quando o programa roda.

NO ENTANTO há uma maneira de um escopo local alterar o escopo global que é através do uso do comando ‘global’, veja:

eggs = 'Hi Lorena'  
*def* spam (*word*) :  
 *global* eggs  
 eggs = *word*spam('Ja acabou, jessica?')  
print(eggs)

Perceba que o que é impresso no console é a frase ‘Já acabou, jessica?’ pois ao declarar que o programa deve tratar a variável ‘eggs’ como sendo global dentro do escopo da função ‘spam()’ ela alterou seu valor global para o valor dado como argumento da função.

EXCEÇÕES/ERROS:

Em python é possível controlar o que o programa deve fazer quando encontra um erro, isto é útil pois é interessante garantir o comportamento correto dele mesmo caso aconteçam erros.

É possível delimitar o comportamento de um erro especifico ou de uma arvore de erros da seguinte maneira:

#Esse erro ocorre ao tentar dividr por zero.  
*def* spam (*divideBy*):  
 *return* 42 / *divideBy  
try*:  
 print(spam(10))  
 print(spam(64))  
 print(spam(0))  
 print(spam(42))  
*except* ZeroDivisionError:  
 print("Argumento da funcao invalido. digite outro valor.")

O bloco ‘try’ tenta executar um código mesmo havendo possibilidade de ocorrer um erro e quando ele ocorre ele aplica a condição respectiva ao erro do bloco exception. No casso este programa imprime os valores da divisão pelos valores dados nos argumentos e quando ele tenta dividr por zero na terceira tentativa o que o programa imprime é o ‘print’ do bloco exception.

C. 4 [p.79] - LISTAS E TUPLAS:

Listas, representadas pelo ‘[]’, e tuplas, representadas pelo ‘()’, são objetos que guardam vários valores, são uteis pois dessa forma podemos condensar e hierarquizar a informação dentro do código de uma maneira mais organizada.

Para criar uma lista basta usar a sintaxe a seguir:

lista2 = ['baiacu','baleia']  
lista1 = ['cachorro','gato','rato',lista2]  
print(lista2[0])  
print(lista1[3])  
print(lista1[3][0])  
print(lista1[-2])

é bem simples, e veja que também é possível acessar listas dentro de listas e valores dessas listas dentro das listas como exemplificado no terceiro print, freezando mais uma vez a importância do uso de listas para organizar o código; também pode-se usar valores negativos para acessar valores dentro da lista, fazendo isso o python interpreta como lendo a lista ao contrario, por exemplo no quarto print ele imprime a palavra rato, pois é o segundo índice de tras para frente.

Slices:

Um conceito também existente quando trabalhando com listas é o de ‘slice’, funciona de maneira análoga ao de seleção de varias células em planilhas, você pega uma range de valores dentro de uma lista, observe:

# slices em listas  
lista = ['oculos','celular','computador','garrafa','sanduiche','cerveja','café']  
print('lista original: {}'.format(lista))  
print('mostrando o slice do 0 ao 3 (lista[:3]): {}'.format(lista[:3]))  
print('mostrnado o slice do 3 ao 6 (lista[3:]): {}'.format(lista[3:]))  
print('mostrando o slice do 2 ao 5 (lista[2:5]): {}'.format(lista[2:5]))

Aqui cabe a observação de que se omitir o primeiro argumento do slice ele leva em conta que vale zero e que se omitir o segundo ele leva em conta que vai ate o fim da lista; Tambem é possível pegar a quantidade de items que tem na lista com o len().

Trabalhando com listas:

Aqui está um exemplo de como trabalhar concatenando strings e listas:

#Listagem de nomes de gatos  
catList = []  
*while True*:  
 print('Digite o nome do gato no. {} ou aperte enter para encerrar o processo.'.format(len(catList)+1))  
 name = input()  
 *if* name == '':  
 *break* catList = catList + [name]  
print('Os nomes dos gatos sao:')  
*for* i *in* range (0, len(catList)): #Aqui criei a variavel 'i' pois quis mostrar o numero do gato dentro da lista no print a seguir.  
 print('O gato no. {} se chama: {}'.format(i+1, catList[i]))  
# caso nao quisesse mostrar poderia ter feito apenas o seguinte:  
*for* name *in* catList:  
 print('Um dos gatos se chama: {}'.format(name))

Quando estudamos loops aprendemos a usar o range(), o código ‘#for i in range(4):...’, por exemplo, só dá certo pois o range(4) é uma função que retorna ao python uma lista [0,1,2,3], por isso é possível fazer o ‘#for name in catList:...’, uma vez que o ‘for’ é um loop que é basicamente usado para contar objetos em uma lista.

In e Not:

Podemos usar o ‘In’ e o ‘Not’ para retornar um valor booleano que dirá se o valor checado está ou não numa lista. Veja:

#Testando In e Not:  
lista = ['cao','gato','periquito']  
spam = 'cao' *in* lista  
print(spam)  
spam = 'baleia' *not in* lista  
print(spam)  
spam = 'macaco' *in* lista  
print(spam)

Macete!

#Macete para adicionar valores contidas em listas a variaveis  
gato = ['gordo','cinza','velha']  
# inves de fazer isso:  
tamanho = gato[0]  
cor = gato[1]  
idade = gato[2]  
#Faça isso:  
tamanho, cor, idade = gato

Métodos:

Segundo o livro: “Método é a mesma coisa que uma função só que é chamada em um valor”, por exemplo:

# Esse programa demonstra um método (index()) aplicado a listas.  
lista = ['baleia','cachorro','gato']  
spam = lista.index('cachorro')  
print(spam)

A variável spam guarda o valor retornado pelo método ‘spam’ e em seguida é impresso.

# exemplo de append(), insert() e remove()

lista = ['baleia','cachorro','gato']

lista.append('macaco')

print(lista)

lista.insert(0,'periquito')

lista.append('gato')

print(lista)

lista.remove('gato')

print(lista)

)

O ‘append()’ insere um item ao final da lista, o ‘insert()’ em uma posição especifica e o ‘remove()’ remove um objeto da lista mas caso existam objetos repetidos é removido apenas o primeiro valor.

Veja o próximo exemplo que utiliza uma lista para armazenar os valores das respostas da bola 8 e então imprime um valor baseado no índice gerado aleatoriamente:

#Bola 8 usando listas  
*import* random  
i = random.randint(0,7)  
bola8 = ["Com certeza sim","Sim","Talvez","Nem pense nisso","Volte mais tarde","Se concentre mais e pergunte novamente","Impossível","Nao"]  
# print(i)  
print(bola8[i])

Tuplas:

Tuplas são praticamente a mesma coisa que listas exceto por duas diferença: 1. São escritas usando ‘()’ invés de ‘[]’ e os valores contidos dentro delas são imutáveis.

Aqui estão as soluções para os projetos do capitulo 4:

#Projeto do capitulo 4  
lista = ['bacon','ovos','salada','feijao']  
print(' e '.join(lista))

C. 5 [p.105] - DICIONÁRIOS:

Dicionários são um tipo de estrutura que assim como listas e tuplas servem para armazenar vario dados, porem invés de índices os valores ficam armazenados atrelados a palavras chave, conhecidas como ‘keys’.

Existem três métodos quando se tratando de dicionários que retornar valores que se assimilam a listas, não se tratam de listas, mas o python interpreta que podem ser usados para loop ‘for’, por exemplo, por serem similares, veja um exemplo de cada:

animais = {'felino': 'gato', 'canino':'cachorro'} #exemplo de dicionario  
  
*for* i *in* animais.values(): #'.value()' retorna os valores dentro de cada key  
 print(i)  
  
*for* j *in* animais.keys(): #'.keys()' retorna os valores das keys dentro do dicionario  
 print(j)  
  
*for* k *in* animais.items(): #'.items()' retorna cada item (keys e respectivos valores) do dict  
 print(k)

OBS: Como dito, esses valores não se tratam de listas mas podem ser transformados em uma usando o ‘lista(\*método do dict\*)’.

Método get():

Este método serve para verificar a existência de um valor key dentro de um dict (dicionário), ele recebe dois argumentos, o primeiro é o valor a ser procurado e o segundo é o valor a ser retornado caso a key pesquisada não seja encontrada. Caso você queira só checar se uma key exista em um dict basta utilizar algo como “print(‘felino’ in animais)” e ele retornará True ou False; veja aseguir um exemplo do get():

mochila = {'caderno':2,'caneta':2,'celular':1} #exemplo de dicionario  
print('Estou levando {caneta} canetas e {maca} maçãs na minha mochila.'.format(caneta = mochila.get('caneta', 0),maca = mochila.get('macas', 0)))

O programa deverá printar:

Estou levando 2 canetas e 0 maçãs na minha mochila.

setdefault():

O setdefault() é um método que checa se já existe uma key(primeiro argumento) com este valor, caso não exista ele cria uma key com este nome e com o valor dado pelo segundo argumento.

Aqui está um exemplo de um programa que conta quantas letras de cada tipo existem em uma string passada:

#Esse programa mostra a quantidade de cada letra em uma dada frase  
text = 'Eu gosto de sorvete de banana'  
count = {}  
*for* character *in* text:  
 count.setdefault(character,0)  
 count[character] += 1  
print(count)

Aqui está o mesmo exemplo só que utilizando o modulo pprint.pprint() que faz com o dicionário seja impresso de uma maneira mais legível:

#Esse programa mostra a quantidade de cada letra em uma dada frase e usa o modulo pprint  
*import* pprint  
text = 'Eu gosto de sorvete de banana'  
count = {}  
*for* character *in* text:  
 count.setdefault(character,0)  
 count[character] += 1  
pprint.pprint(count)

Segue um exemplo de um jogo da velha feito utilizando um exemplo de estruturação de dados para modelar um tabuleiro de jogo da velha usando um dicionário; nesse exemplo cada casa do tabuleiro é atribuída a uma key no dicionário criado como tabuleiro e o usuário pode modificar esse valor entrando com a casa que ele deseja colocar a pedra:

#Jogo da velha / tic-tac-toe  
theBoard = {  
 'top-L':' ','top-M':' ','top-R':' ',  
 'mid-L':' ','mid-M':' ','mid-R':' ',  
 'low-L':' ','low-M':' ','low-R':' '  
}  
*def* printBoard(*board*):  
 print('{}|{}|{}'.format(*board*['top-L'],*board*['top-M'],*board*['top-R']))  
 print('-+-+-')  
 print('{}|{}|{}'.format(*board*['mid-L'],*board*['mid-M'],*board*['mid-R']))  
 print('-+-+-')  
 print('{}|{}|{}'.format(*board*['low-L'],*board*['low-M'],*board*['low-R']))  
  
turn = 'X'  
*for* i *in* range(9):  
 printBoard(theBoard)  
 print('Vez de: ' + turn + '. Qual espaço deseja colocar sua peça? ')  
 move = input()  
 theBoard[move] = turn  
 *if* turn == 'X':  
 turn = 'O'  
 *else*:  
 turn = 'X'

Também é possível, inclusive muito útil em algumas situações, utilizar dicionários dentro de outros dicionários; veja no exemplo a seguir como saber o total de itens levados em uma festa:

#exemplo de dicionarios dentro de dicionarios  
convidados = {'Alice' : {'Bolos' : 1, 'Refrigerantes' : 2, 'Baloes' : 3},  
 'Antonia' : {'Salgadinhos' : 1, 'Sucos' : 2, 'Bolos' : 3},  
 'Elis' : {'Bolos' : 1, 'Salgadinhos' : 2, 'Sucos' : 3}}  
  
*def* totalItens(*convidados*,*item*): #Retorna o total do item discriminado  
 trazidos = 0  
 *for* i, k *in convidados*.items():  
 trazidos = trazidos + k.get(*item*, 0)  
 *return* print('- Numero de {} para a festa: {}'.format(*item*, trazidos))  
  
totalItens(convidados,'Bolos')  
totalItens(convidados,'Refrigerantes')  
totalItens(convidados,'Baloes')  
totalItens(convidados,'Sucos')

Perceba que neste exemplo, dentro do loop for, o valor de ‘i’ é associado ao nome da pessoa e o valor de ‘k’ é associado ao dicionário que contem os itens da festa (no caso o segundo valor dentro de cada item do dicionário mais externo); então se usa o ‘get()’ para pegar, caso exista, o valor associado ao item escolhido no segundo argumento da função.

Esta notação com duas variáveis no ‘for’ é muito útil quando se tratando de dicionários, veja no exemplo a seguir mais uma demonstração dessa notação, agora utilizando o projeto proposto no final do capitulo pelo livro:

#Itens de um inventario de RPG  
stuff = {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12} #Itens do inventario  
  
*def* displayInventory(*inventory*):#Funcao que retorna todos os itens do inventario e suas quantidades  
 print("Inventory:")  
 item\_total = 0  
 *for* itemName, numberOfItens *in inventory*.items():  
 print('- {} {}'.format(numberOfItens, itemName))  
 item\_total += v  
 print("Total number of items: " + str(item\_total))  
  
displayInventory(stuff)#chamada da funcao

Aqui o ‘for’ funciona como no exemplo anterior mas de uma maneira mais direta pois não contem dicionários dentro do dicionário; perceba que o primeiro valor a ser iterado(itemName) no loop ‘for’ se trata do ‘key’ dentro do dicionário, e o segundo (numberOfItens) é o ‘value’ associada àquela ‘key’.

Aqui está a continuação do exemplo agora com uma função de adicionar itens que vem a partir de uma lista:

#Itens de um inventario de RPG  
stuff = {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12} #Itens do inventario  
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']#Itens para ser adicionados  
*def* displayInventory(*inventory*):#Funcao que retorna todos os itens do inventario e suas quantidades  
 print("Inventory:")  
 item\_total = 0  
 *for* itemName, numberOfItens *in inventory*.items():  
 print('- {} {}'.format(numberOfItens, itemName))  
 item\_total += numberOfItens  
 print("Total number of items: " + str(item\_total))  
  
*def* addToinventory(*inventory*, *addedItems*):#Funcao que adiciona os itens de uma lista no dicionario utilizado como inventario  
 *for* item *in addedItems*:  
 *if* item *in inventory*:  
 *inventory*[item] += 1  
 *inventory*.setdefault(item ,1)  
 print('New itens added to inventory!'.format(*addedItems*))  
  
addToinventory(stuff, dragonLoot)  
displayInventory(stuff)

E aqui está a minha primeira tentavia no programa sem a ajuda do livro:

#Meu codigo  
bigBag = {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12} #inventario  
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby'] # Itens para ser adicionados  
  
*def* showInventory(*inventory*):  
 totalOfItems = 0  
 *for* i *in inventory*.items():  
 print('{} {}'.format(i[1], i[0]))  
 totalOfItems += i[1]  
 print('Total number of items: {}'.format(totalOfItems))  
  
*def* addItems(*inventory*, *newItems*):  
 *for* item *in newItems*:  
 *if* item *in inventory*:  
 *inventory*[item] += 1  
 *inventory*.setdefault(item, 1)  
 print('New items added to inventory!' )  
  
addItems(bigBag,dragonLoot)  
showInventory(bigBag)

C. 6 [p.123] - MANIPULANDO STRINGS:

Como já estudado, strings são um dataType em python que consiste numa sequencia determinada de caracteres para formar frases ou coisas do gênero.

\ (barra invertida) e aspas triplas:

A barra é utilizada para que o python possa exibir caracteres que de outra forma não seria exibida, veja os exemplo na tabela abaixo:

Símbolo Print:

\' Aspas simples

\" Aspas duplas

\t Tab

\n Nova linha (line break)

\\ Barra invertida

# Exemplo de simbolos e aspas triplas  
print(''''Para: Leo  
Bom dia, Leo.  
Hoje o dia será chuvoso,   
Um certo \'Alguém\' Baterá na sua porta, NÃO ATENDA!  
Assinado: Leo do futuro.  
''')

Note aqui que foi usado ‘\’’ apenas para demonstrar como funciona pois quando se usa a aspas triplas não é necessário.

Índices em strings:

É possível usar índices também, veja:

# Indices em Strings  
spam = 'Hello World!'  
  
print('spam[1]: {}'.format(spam[1]))  
print('spam[3]: {}'.format(spam[3]))  
print('spam[-6]: {}'.format(spam[-6]))  
  
print('spam[1:6]: {}'.format(spam[1:6]))

In e not in:

spam = 'Meu nome é Bleblio'  
  
print('Bleblio' *in* spam)  
print('é' *not in* spam)

Métodos Strings úteis:

Aqui temos os exemplos de ‘upper()’ e ‘lower()’ que funcionam como o nome sugere:

#upper() e lower()  
spam = 'Walk like an Egyption....'  
  
print(spam)  
spam = spam.lower()  
print(spam)  
spam = spam.upper()  
print(spam)

Também tem-se os métodos isupper() e o islower() que checa se dado texto/variável está capitializado ou não e retorna um valor booleano. Há ainda vários outros métodos que retornam um booleano e checam para várias outras coisas, veja:

#Métodos isX()  
  
spam = 'Testando...'  
print(spam.isalpha()) #returns True if the string consists only of letters and is not blank.  
  
spam = '963852741 Tenho numeros também!'  
print(spam.isalnum()) #returns True if the string consists only of letters and numbers and is not blank.  
  
spam = '998855442211'  
print(spam.isdecimal()) #returns True if the string consists only of numeric characters and is not blank.  
  
spam = ' \n'  
print(spam.isspace()) #returns True if the string consists only of spaces, tabs, and newlines and is not blank.  
  
spam = 'Título'  
print(spam.istitle()) #returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

Estes métodos a princípio parecem bobos, mas são bem uteis quando necessitamos validar inputs de usuário:

# Exemplo de uso de método isX()  
*while True*:  
 idade = input('Digite sua idade: ')  
 *if* idade.isdecimal():  
 *break* print('Digite um número para sua idade!')  
*while True*:  
 nome = input('Digite seu nome: ')  
 *if* nome.isalpha():  
 *break* print('Digite um nome somente com letras!')  
print('Bem vindo {}'.format(nome))

Veja mais dois exemplos de outros dois métodos:

# starswith() e endswith()  
spam = 'Hello World!'  
print(spam.startswith('Bola'))  
print(spam.endswith('ld!'))

Split() e join():

São dois métodos que são utilizado para obter listas a partir de uma string (split()) e para obter um string a partir de uma lista (join()), veja:

# split() e join()  
spam = 'Estou cansado de sempre ouvir Tarkus, porque demora tanto pra acabar?!'  
  
spam = spam.split()  
print(spam)  
  
spam = ' '.join(spam)  
print(spam)

Justificando textos:

O Python também oferece três maneiras de justificar textos:  
ljust(), rjust() e o center().

#Exemplos de justificacao de texto  
spam = 'Bolinha de Golfe'  
print(spam.ljust(25,'='))  
print(spam.rjust(25,'\*'))  
print(spam.center(25,'+'))

O primeiro argumento de cada método significa quantos caracteres eu quero encaixar o texto e o segundo o caractere que eu quero ele encaixado, o segundo é opcional, sendo que caso seja omitido o texto é justificado com ‘ ’.

Removendo espaços em brando:

O strip(), lstrip() e rstrip() remove caracteres específicos de uma string desde que estes estejam em seus extremos:

#Remove caracteres  
spam = '++++Olá mundo!++++'  
print(spam.strip('+'))  
print(spam.lstrip('+'))  
print(spam.rstrip('+'))  
  
spam = 'sabaooooooooooooo'  
print(spam.strip('o'))

Perceba que neste exemplo, no segundo ‘spam’ é removido todos os ‘o’ da string.

Copiando e Colando strings:

Existe ainda o modulo pyperclip que pode ser baixado através do comando no powershell ‘pip install’, ele permite ao usuário copiar e colar strings de dentro para fora do python e vice, versa, veja:

#pyperclip demo  
*import* pyperclip  
pyperclip.copy('Olá Mundo!') #o que estiver como argumento será o texto enviado para a area de copia do usuario  
spam = pyperclip.paste() #a variavel 'spam' recebe o valor que está na area de copia do usuario  
print(spam)

C.7 [p.147] – EXPRESSÕES REGULARES (REGEX):

A partir daqui comecei a usar o Visual Code Studio então os códigos estarão com uma formatação um pouco diferente.

Em Python é possível procurar expressões por padrões, como por exemplo padrões de números de telefone. Isso é especialmente útil quando queremos extrair certos tipos de informações de algum bloco de texto ou coisa parecida. Veja no exemplo a seguir como confirmar se um número de telefone (americano) é válido ou não:

def isPhoneNumber(text):

    if len(text) != 12:

        return False

    for i in range(0,3):

        if not text[i].isdecimal():

            return False

    if text[3] != '-':

        return False

    for i in range(4,7):

        if not text[i].isdecimal():

            return False

    if text[7] != '-':

        return False

    for i in range(8, 12):

        if not text[i].isdecimal():

            return False

    return True

print('415-555-4242 is a phone number:')

print(isPhoneNumber('415-555-4242')) #True

print('Moshi moshi is a phone number:')

print(isPhoneNumber('Moshi moshi')) #False

Este programa usa um principio de ‘peneiramento’, se ele passar por todas as condições que validam um número de telefone como verdadeiro ele retorna True, caso contrario retorna False; Se quiséssemos checar se existe algum número de telefone em um bloco de texto podemos trabalhar com ele da seguinte maneira:

def isPhoneNumber(text):

    ...

def findPN(text):

    for i in range(len(text)):

        chunk = text[i:i+12]

        if isPhoneNumber(chunk):

            print('Number found: {}'.format(chunk))

message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'

findPN(message)

Perceba que para escrever esse código, pegamos as características intrínsecas que um numero de telefone americano deve ter e transformamos em regras que são peneiradas pela função ‘isPhoneNumber()’ e então em ‘findPN()’ criamos um loop que lê de 12 em 12 caracteres (o comprimento de um número de telefone) e checa as condições definidas em ‘isPhoneNumber()’

Regexes ou padrões de texto:

Regex é uma maneira de escrever código que procura por padrões de texto usando uma notação sucinta e útil. Por exemplo, ‘\d’ é utilizado em regex para delimitar qualquer caractere numeral (0-9) sozinho, sendo assim, é possível delimitar qualquer numero de telefone utilizando a expressão ‘\d\d\d-\d\d\d-\d\d\d\d’.

Para que se possa utilizar as expressões regex para encontrar determinados padrões de caracteres primeiros precisamos de um objeto regex e para criar uma fazemos:

import re #importando

phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') #objeto regex

                          #^ 'r' serve para a strign que segue nao utilizar escape characters

spam = phoneNumRegex.search('My number is 415-555-4242.')

print('Phone found: ' + spam.group())

E para listar todos os valores encontrados no bloco de texto passado podermos fazer o seguinte:

import re #importando

phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') #objeto regex

                          #^ 'r' serve para a strign que segue nao utilizar escape characters

spam = phoneNumRegex.findall('Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.')

print(spam) # ['415-555-1011', '415-555-9999']

A lista de métodos incluído neste modulo está aqui: <https://docs.python.org/3/library/re.html>

Grupos de caracteres:

Utilizando Regex é possível criar grupos para diferenciar padrões. Primeiramente deve-se criar um objeto regex da mesma maneira que no código anterior porem colocando um parênteses em um grupo de caracteres especifico que se queira delimitar, então ao printar ou fazer qualquer operação com o MO (match object) utilizar mo.group(n) em que ‘n’ é o numero do grupo que se dessa acessar (caso deixado em branco ele retorna todos os grupo concatenados), veja:

#regex - group()

import re

phoneNumberRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)') #separam-se os grupos utilizando parenteses

mo = phoneNumberRegex.search('415-555-4242 is a phone number') #criando o MO

print(mo.group(1)) #415

print(mo.group(2)) #555-4242

print(mo.group( )) #415-555-4242

Há ainda a alternativa de utilizar o ‘groups()’ que retorna uma tupla com os dados captados:

#regex - groups()

import re

phoneNumberRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)') #separam-se os grupos utilizando parenteses

mo = phoneNumberRegex.search('415-555-4242 is a phone number') #cria-se o MO

areaCode, mainNumber = mo.groups()

print('Tuple (mo.groups()): {}'.format(mo.groups())) #('415', '555-4242')

print('Area code (group 1): ' + areaCode)

print('Main number (group 2): ' + mainNumber)

Perceba que neste caso utilizamos o truque de atribuir um array à duas variáveis, o que faz com que o primeiro valor do array(tupla) seja atribuída à primeira variável da sequência, e o segundo à segunda e assim por diante...

A partir daqui vamos ver alguns caracteres úteis em regex, vale lembrar que caso queira utilizar o caractere em si para ser buscado é necessário o uso do ‘escape character’ [\]

É possível utilizar o símbolo ‘|’ para determinar que um regex pegue um conjunto ou outro de caracteres:

#regex - pipe [|]

import re

batRegex = re.compile(r'(Bat|bat)(man|mobile|copter|tery)')

mo = batRegex.findall('Look! Batman just got off the batcopter and it\'s entering the batmobile but he doesn\'t know it\'s out of battery!')

print(mo) # [('Bat', 'man'), ('bat', 'copter'), ('bat', 'mobile'), ('bat', 'tery')]

Neste exemplo o código pega um conjunto de [‘Bat’ ou ‘bat’]+[ ‘man’ ou ‘mobile’ ou ‘copter’ ou ‘tery’]

No próximo exemplo podemos ver um exemplo de como utilizar um grupo de caracteres que podem ou não fazer parte da string analizada:

#regex - question mark [?]

import re

batRegex = re.compile(r'Bat(wo)?man') #re.compile para o padrao que estou buscando

mo = batRegex.search('Look! it\'s the Batwoman!') #variavel.seacrh('texto')|findall('texto') para enconstrar no texto o padrao compilado

print(mo.group()) #Batwoman

Utilizando o [\*] é possível encontrar qualquer combinação de caracteres que aconteça zero ou mais vezes em um texto:

#regex - asterisk [\*]

import re

batRegex = re.compile(r'Bat(wo)\*man')

mo = batRegex.search('The adventures of Batwowowowowowowoman')

print(mo.group()) #Batwowowowowowowoman

O mesmo vale para o [+] mas significa um ou mais vezes:

#regex - plus sign [+]

import re

batRegex = re.compile(r'Bat(wo)+man')

mo\_1 = batRegex.search('The adventures of Batwowowoman')

mo\_2 = batRegex.search('The adventures of Batman')

print(mo\_1.group()) #Batwowowoman

print(mo\_2 == None) #True

Aqui a expressão ‘mo\_2 == None’ é igual a ‘True’ pois já que o código não encontrou nem uma vez seque o ‘wo’ no segundo texto mo\_2 retorna o valor ‘None’.

Caso queira procurar por algo em repetição, digamos o bloco ‘Ha’ um determinado número de vezes ou em range, deve-se utilizar o ‘{}’

#regex - curly brackets [{}]

import re

haRegex\_strict3 = re.compile(r'(Ha){3}') #match strictly 3 times 'Ha'

haRegex\_3to5 = re.compile(r'(Ha){3,5}') #match between 3 and 5 times 'Ha'

haRegex\_min3 = re.compile(r'(Ha){3,}') #match at least 3 times 'Ha'

haRegex\_max5 = re.compile(r'(Ha){,5}') #match at max 5 times 'Ha'

mo\_strict3 = haRegex\_strict3.search('HaHaHaHaHaHa')

mo\_3to5 = haRegex\_3to5.search('HaHaHaHaHaHa')

mo\_min3 = haRegex\_min3.search('HaHaHaHaHaHa')

mo\_max5 = haRegex\_max5.search('HaHaHaHaHaHa')

print(mo\_strict3.group()) #HaHaHa (3 times)

print(mo\_3to5.group()) #HaHaHaHaHa (5 times)

print(mo\_min3.group()) #HaHaHaHaHaHa (6 times)

print(mo\_max5.group()) #HaHaHaHaHa (5 times)

Como visto anteriormente, Python tende a ser ganancioso quando utilizando regex e tenta pegar o maior numero de objetos procurados possíveis, por isso no exemplo anterior sempre retorna o máximo de vezes; porém utilizando o já conhecido ‘?’ é possível fazer o sistema entender que queremos pegar o mínimo:

#regex - greedy vs nongreedy

import re

greedyRegex = re.compile(r'(Ha){3,5}')

mo1 = greedyRegex.search('HaHaHaHaHaHa')

mo1.group()

print(mo1)

nonGreedyRegex = re.compile(r'(Ha){3,5}?')

mo2 = nonGreedyRegex.search('HaHaHaHaHaHa')

mo2.group()

print(mo2)

findall():

O método ‘search()’ pega o primeiro elemento igual ao regex passado para ele, já o ‘findall()’ como o nome sugere, retorna todos os elementos dentro da string que equivalem ao regex solicitado em forma de lista:

#regex - findall()

import re

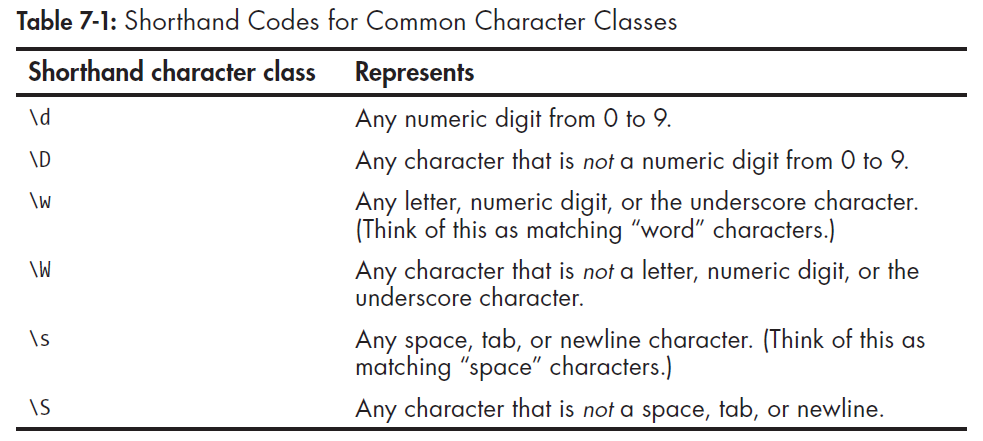
phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')

mo = phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')

print(type(mo)) #list

print(mo) #['415-555-9999', '212-555-0000']

A seguir esta uma tabela que mostra os tipos de caracteres e seu códigos correspondentes em regex:



Exemplo de como filtrar uma lista de itens de natal:

#regex - xmas list using regex

import re

xmasRegex = re.compile(r'\d+\s\w+')

xmaslist = xmasRegex.findall('1 doll, 2 pens, 1 coal, 2 candy and 1 truck')

print(xmaslist) #['1 doll', '2 pens', '1 coal', '2 candy', '1 truck']

Existem momentos em que as classes de caracteres existentes (\w, \d, ...) se tornam muito genéricos e não são suficientes para fazer o que se quer, para contornar este problema em Python é possível criar suas próprias classes de caracteres:

# regex - making character classes

import re

vowelRegex = re.compile(r'[aeiouAEIOU]')

text = 'I like to write code in python because I can make cool stuff'

mo = vowelRegex.findall(text)

print(mo) #['I', 'i', 'e', 'o', 'i', 'e', 'o', 'e', 'i', 'o', 'e', 'a', 'u', 'e', 'I', 'a', 'a', 'e', 'o', 'o', 'u']

Veja também um exemplo de como utilizar o ‘^’ que é usado para excluir caracteres e o hífen ‘-’ que trabalha com uma range de caracteres:

# regex - character classes properties

import re

consonantsRegex = re.compile(r'[^aeiouAEIOU ]')

oneToFiveRegex = re.compile(r'[1-5]')

text = 'Robocop likes to eat baby food at least 3 times per day'

mo1 = consonantsRegex.findall(text)

mo2 = oneToFiveRegex.findall(text)

print(mo1)#['R', 'b', 'c', 'p', 'l', 'k', 's', 't', 't', 'b', 'b', 'y', 'f', 'd', 't', 'l', 's', 't', '3', 't', 'm', 's', 'p', 'r', 'd', 'y']

print(mo2)#['3']

Para encontrar objetos que se encaixem me padrões que comecem ou terminem com determinado set de caracteres bastam usar o ^ e $ respectivamente:

#regex - how to set beggining and ending of a strign

import re

begginingRegex = re.compile(r'^Hello')

endingRegex = re.compile(r'\d$')

text0 = 'Robocop is a bad movie'

text1 = 'Hello world!'

text2 = 'The correct answer is 42'

mo01 =begginingRegex.findall(text0)

mo02 = endingRegex.findall(text0)

mo1 = begginingRegex.findall(text1)

mo2 = endingRegex.findall(text2)

print(mo01) #[] – None found

print(mo02) #[] – None found

print(mo1) #['Hello']

print(mo2) #['2']

Para encontrar qualquer caractere utilizamos o [.], também chamado de ‘wildcard’:

#regex - wildcard [.]

import re

wildcardRegex = re.compile(r'.at')

mo = wildcardRegex.findall('The cat in the hat sat on the flat mat.')

print(mo) #['cat', 'hat', 'sat', 'lat', 'mat']

Para encontrar tudo utiliza-se o [.\*] que significa ‘qualquer caractere que apareça uma ou mais vezes’:

#regex - matching anything

import re

namesRegex = re.compile(r'First Name: (.\*) Last Name: (.\*)')

text = 'First Name: Leonardo Last Name: Machado'

mo = namesRegex.findall(text)

print(mo) #[('Leonardo', 'Machado')]

OBS: caso queira que o programa encontre apenas as letras, sem se importar com o sensitive case (se estará maiúscula ou minúscula) deve-se passar o [re.IGNORECASE ou re.I] como segundo argumento dentro de re.compile()

O método sub() possibilita substituir por uma determinada string o que for encontrado pelo regex criado:

#regex - sub() method

import re

subRegex = re.compile(r'Agent \w+')

text = 'Agent Alice turned the papers to Agent Bob last week'

sub = subRegex.sub('CENSORED', text)

print(sub) #CENSORED turned the papers to CENSORED last week

Com este conhecimento sobre regex em mente, temos uma manha para quando tentarmos criar regex muito grandes e complicados: é possível utilizar o re.VERBOSE como segundo argumento de re.compile() para que este ignore espaços em branco, newLines e comentários, possibilitando uma maneira mai fácil de interpretar os regex que estamos trabalhando :

#regex - re.VERBOSE

import re

phoneRegex1 = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}(\s\*(ext|x|ext.)\s\*\d{2,5})?)')

phoneRegex2 = re.compile(r'''(

(\d{3}|\(\d{3}\))?              # area code

(\s|-|\.)?                      # separator

\d{3}                           # first 3 digits

(\s|-|\.)                       # separator

\d{4}                           # last 4 digits

(\s\*(ext|x|ext.)\s\*\d{2,5})?    # extension

)''', re.VERBOSE)

E se o projeto pedir que se utilize 2 ou mais métodos de formatação do re.compile()? Basta utilizar o caractere [|] no segundo argumento:

import re

someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)

PROJETO:

A seguir mostra-se como utilizar regex e o pyperclip para poder encontrar informações de contato que foram copiadas para o clipboard:

import re, pyperclip

def getContact():

    #Get clipboard info

    text = str(pyperclip.paste())

    #Regex

    emailRegex = re.compile(r'''(

    [a-zA-Z0-9.\_+-]+                # username

    @                               # @

    [a-zA-Z0-9]+                    # domain

    \.[a-zA-Z]+                     # .something

    )''',re.VERBOSE)

    phoneRegex = re.compile(r'''(

    (\+\d{2})?                      # country

    (\d{3}|\(\d{3}\))?              # area code

    (\s|-|\.)?                      # separator

    \d{3}                           # first 3 digits

    (\s|-|\.)                       # separator

    \d{4}                           # last 4 digits

    (\s\*(ext|x|ext.)\s\*\d{2,5})?    # extension

    )''', re.VERBOSE)

    #Get email and phone number list

    bufferList = phoneRegex.findall(text)

    emailList = emailRegex.findall(text)

    phoneList = []

    for i in range(len(bufferList)):

        phoneList.append(bufferList[i][0])

    #Format lists to str

    resultString = ''

    for i in range(len(emailList)):

        resultString += ('\n{}'.format(emailList[i]))

    resultString += '\n'

    for i in range(len(phoneList)):

        resultString += ('\n{}'.format(phoneList[i]))

    #Result

    print('\nThe following info was copied to your clipboard: \n {}'.format(resultString))

    return  pyperclip.copy(resultString)

getContact()

C.8 [p.173] – MANIPULANDO ARQUIVOS:

Um arquivo é constituído de dois parâmetros primário: Seu nome e seu caminho.

A primeira propriedade, geralmente constituído de apenas uma palavra, denota como o arquivo é chamado, já a segunda denota onde o mesmo se localiza.

Quanto ao nome não existem muitas diferenças significativas entre um OS e outro, porém quanto ao caminho existem sim.

No Windows, utiliza-se a notação C:\, em quanto em Mac e Linux utiliza-se C:/. Para denotar o *root* do computador.

Isso influencia na maneira como escrevemos um programa, pois escrever um código que faz um arquivo .txt ser gravado em “C:\leonardo\txt.txt” é diferente de escrever “C:/.leonardo/text.txt”. Para contornar este problema utilizamos o método ‘*os.path.join('usr', 'bin', 'spam')*’ que vem do **import os** que retorna uma string do caminho a ser usado, neste caso, 'usr\\bin\\spam' pois foi compilado em Windows.

#OS path example

import os

files = ['contacts.xls','welcome.txt','accounts.xls']

for file in files:

    print(os.path.join('C:\\users\\leonardo\\documents', file))

    #Prints:

    # C:\users\leonardo\documents\contacts.xls

    # C:\users\leonardo\documents\welcome.txt

    # C:\users\leonardo\documents\accounts.xls

OBS: usa-se [\\] pois é necessário o uso do escape character; de maneira análoga, pode-se utilizar o [(r'C:\users\leonardo\documents', file)]

Adicionalmente, todos os programas rodam em um dire’torio específico, ou *cwd* (*current working directory*), os dois comando mostrados a seguir interagem com o cwd:

# CWD -> Current Working Directory

import os

print(os.getcwd()) #C:\Users\Leonardo\Projects\Python

#os.chdir('C:\\Users\\Leonardo\\thisPathDoesntExist') #FileNotFoundError

os.chdir('C:\\Users\\Leonardo\\Desktop')

print(os.getcwd()) #C:\Users\Leonardo\Desktop

Também é importante conhecer a noção de caminho absoluto e relativo; como os nomes sugerem, o primeiro sugere relação com o *root* [C:] e o segundo com o *cwd*.

O [.\] se refere a um arquivo/diretório filho em relação ao *cwd* e [..\] que se refere a um diretório pai; por exemplo:

* Se estivermos em [C:\users\leonardo], [.\pastel.txt] está em [C:\users\leonardo\pastel.txt];
* Agora se estivermos em [C:\users\leonardo], [..\pastel.txt] está em [C:\users\pastel.txt].

Essas funções servem para validar caminhos relativos e absolutos:

* [os.path.abspath(path)]: Retorna uma string com o caminho absoluto do argumento;
* [os.path.isabs(path)]: Retorna True se for um caminho absoluto e False caso não;
* [os.path.relpath(path, start)]: Retorna um caminho relativo entre o *‘path’* e o *‘start’*, caso o *‘start’* seja omitido é utilizado o *cwd*.

#Absolute and relative paths

import os

os.path.abspath('.\\') #c:/Users/Leonardo/Projects/Python/chapter8\_files.py

os.path.isabs('c:/Users/Leonardo/Projects/Python/chapter8\_files.py') #True

os.path.relpath('C:\\Users\\Leonardo\\Desktop') #'..\\..\\..\\..\\..\\Desktop'

CRIANDO DIRETÓRIOS:

É possível criar diretórios utilizando o modulo *os*, toda a documentação se encontra em <https://docs.python.org/3/library/os.path.html>, veja:

#Creating directories

import os

os.makedirs('C:\\Users\\Leonardo\\Desktop\\test\\spam\\eggs\\foo')

Neste caso o programa criou os diretórios [test], [spam], [eggs] e [foo].

INFORMAÇÕES ÚTEIS DOS DIRETÓRIOS:

Aqui estão demonstradas algumas maneiras de obter algumas informações úteis sobre os diretórios e seus caminhos:

#getting Dir info

import os

path = 'C:\\Windows\\System32\\calc.exe'

os.path.basename(path) #'calc.exe'

os.path.dirname(path) #'C:\\Windows\\System32'

os.path.split(path) #('C:\\Windows\\System32', 'calc.exe')

calcPath = 'C:\\Windows\\System32\\calc.exe'

calcPath.split(os.path.sep) #['C:', 'Windows', 'System32', 'calc.exe']

OBS: Aqui foi utilizado o ‘*cmd*’ para poder executar os códigos.

ARQUIVOS: TAMANHOS E CONTEÚDOS: