Curso Rápido de Python

Formatação de espaços em branco

Identação como delimitação de bloco

In [ ]:
```
for i in [1, 2, 3, 4, 5]:
    print(i) # first line in "for i" block
    for j in [1, 2, 3, 4, 5]:
        print(j) # first line in "for j" block
        print(i + j) # last line in "for j" block
    print(i) # last line in "for i" block
print("done looping")
```

Espaços em braco são ignorados em parênteses e colchetes

In [ ]:
```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 +
12 +
13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)

list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

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

In [ ]:
```
print(easier_to_read_list_of_lists)
```

Uso de barra invertida para informar que a sentença continua na próxima linha

In [ ]:
```
two_plus_three = 2 + \
                 3
```

In [ ]:
```
two_plus_three
```

Módulos

In [ ]:
```
import re
my_regex = re.compile("[0-9]+", re.I)
```

In [ ]:
```
print(my_regex.match("71abc").group())
```

In [ ]:
```
import matplotlib.pyplot as plt
```

In [ ]:
```

```

In [ ]:
```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

In [ ]:
```
match = 10
from re import * # uh oh, re has a match function
print(match) # "<function re.match>"
```

Aritmética

Python 2.7 usa divisão inteira por padrão, deste modo 5 / 2 = 2.
Para obtermos um resultado fracionado temos que incluir o seguinte import:

In [ ]:
```
from __future__ import division
```

Para obtermos o resultado inteiro usamos duas barras, como a seguir:

In [ ]:
```
int_res = 5 // 2
```

In [ ]:
```
int_res
```

Funções

Em Python definimos funções usando DEF

In [ ]:
```
def double(x):
    """Comentário de várias linhas
    para explicar o que a função faz.
    Por exemplo, esta função multiplica o valor de entrada por 2"""
    return x * 2
```

Podemos atribuir funções à variáveis e passá-las como argumentos de outras funções.

In [ ]:
```
def apply_to_one(f):
    """chama a função f passando 1 como argumento"""
    return f(1)

my_double = double          # refers to the previously defined function
x = apply_to_one(my_double) # equals 2

print(x)
```

também é fácil de criar pequenas funções anônimas ou lambdas, como são conhecidas

In [ ]:
```
y = apply_to_one(lambda x: x + 4) # equals 5

print(y)
```

In [ ]:
```
def add(x, y):
    return x + y
```

In [ ]:
```
add(2, 3)
```

In [ ]:
```
add = lambda x, y: x + y
```

In [ ]:
```
add(3, 4)
```

In [ ]:
```
import math
```

In [ ]:
```
sorted([1, 2, 3, 4, 5], key=lambda x: x, reverse=True)
```

In [ ]:
```
sorted([1, 2, 3, 4, 5], key=lambda x: math.log(x), reverse=True)
```

In [ ]:
```
map(lambda x: x * x, [1,2,3,4,5])
```

Pârametros de funções podem ter valor padrão

In [ ]:
```
def my_print(message="my default message"):
    print(message)

my_print("hello") # prints 'hello'
my_print() # prints 'my default message'
```

In [ ]:
```
def subtract(a=0, b=0):
    return a - b

subtract(10, 5) # returns 5
subtract(0, 5) # returns -5
subtract(b=5) # same as previous
```

Strings

Podem ser delimitadas por aspas " ou apóstrofos ', mas tem que combinar.

In [ ]:
```
single_quoted_string = 'data science'
double_quoted_string = "data science"
```

Python usa barra invertida para codificar caracteres especiais.

In [ ]:
```
tab_string = "\t" # represents the tab character
len(tab_string) # is 1
```

Para se obter uma barra invertida como caractere, deve-se criar uma string usando o R""

In [ ]:
```
not_tab_string = r"\t" # represents the characters '\' and 't'
len(not_tab_string) # is 2
print(not_tab_string)
```

Para se criar uma string de múltiplas linhas, deve-se usar triplo-aspas (""")

In [ ]:
```
multi_line_string = """This is the first line.
and this is the second line
and this is the third line"""
```

Exceções

Quando algo dá errado Python dispara uma excecão que, se não tratada, pode encerrar o programa.

In [ ]:
```
try:
    print(3 / 0)
except ZeroDivisionError:
    print("cannot divide by zero")
```

In [ ]:
```
try:
    print(3 / 1)
except Exception as err:
    print(err)
    print("cannot divide by zero")
else:
    print('Executing the else clause.')
finally:
    print('Cleaning up, irrespective of any exceptions.')
```

Listas

A lista é provavelmente a estrutura mais fundamental em Python, ela é similar a um array porém com algumas funcionalidades

In [ ]:
```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [ integer_list, heterogeneous_list, [] ]

list_length = len(integer_list) # equals 3
print(list_length)

list_sum = sum(integer_list) # equals 6
print(list_sum)
```

Você pode obter ou atribuir o enésimo elemento de uma lista usando colchetes

In [ ]:
```
x = list(range(10)) # is the list [0, 1, ..., 9]
zero = x[0] # equals 0, lists are 0-indexed
one = x[1] # equals 1
nine = x[-1] # equals 9, 'Pythonic' for last element
eight = x[-2] # equals 8, 'Pythonic' for next-to-last element
x[0] = -1 # now x is [-1, 1, 2, 3, ..., 9]
```

Também usamos colchetes para fatiar as listas

In [ ]:
```
first_three = x[:3] # [-1, 1, 2]
three_to_end = x[3:] # [3, 4, ..., 9]
one_to_four = x[1:5] # [1, 2, 3, 4]
last_three = x[-3:] # [7, 8, 9]
without_first_and_last = x[1:-1] # [1, 2, ..., 8]
copy_of_x = x[:] # [-1, 1, 2, ..., 9]
```

Python possui o operador "in" que verifica se um elemento é membro de uma lista

In [ ]:
```
1 in [1, 2, 3] # True
0 in [1, 2, 3] # False
```

É fácil concatenar listas

In [ ]:
```
x = [1, 2, 3]
print(x)

x.extend([4, 5, 6]) # x is now [1,2,3,4,5,6]
print(x)
```

Para incluir novos elementos em "x" sem modificá-lo, faça:

In [ ]:
```
x = [1, 2, 3]
y = x + [4, 5, 6] # y is [1, 2, 3, 4, 5, 6]; x is unchanged
```

É comum incluir um item novo de cada vez a uma lista

In [ ]:
```
x = [1, 2, 3]
x.append(0) # x is now [1, 2, 3, 0]
y = x[-1] # equals 0
z = len(x) # equals 4
```

Também podemos desempacotar uma lista

In [ ]:
```
x, y = [1, 2] # now x is 1, y is 2
print(x, y)
```

Ao desempacotar, se não quisermos um elemento da lista podemos usar um underscore (_)

In [ ]:
```
_, y = [1, 2] # now y == 2, didn't care about the first element
print(y)
```

Tuplas

Tuplas são primos imutáveis das listas.
Você especifica uma tupla usando parênteses em vez de colchetes.

In [ ]:
```
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3 # my_list is now [1, 3]

try:
    my_tuple[1] = 3
except TypeError:
    print("cannot modify a tuple")
```

Tuplas são um modo conveniente de retornar múltiplos valores de uma função

In [ ]:
```
def sum_and_product(x, y):
    return (x + y),(x * y)

sp = sum_and_product(2, 3) # equals (5, 6)
print(sp)

s, p = sum_and_product(5, 10) # s is 15, p is 50
print(s, p)
```

Tuplas (e listas) podem ser usadas para atribuições múltiplas

In [ ]:
```
x, y = 1, 2 # now x is 1, y is 2
print(x, y)

x, y = y, x # Pythonic way to swap variables; now x is 2, y is 1
print(x, y)
```

Dicionários (Dicts)

Outra estrutura de dados fundamental em Python.
Associa chaves a valores e permite rápido acesso aos dados

In [ ]:
```
empty_dict = {} # Pythonic
empty_dict2 = dict() # less Pythonic
grades = { "Joel" : 80, "Tim" : 95 } # dictionary literal
print(grades)
```

Buscando valores usando chave

In [ ]:
```
print(grades['Joel'])

try:
    kates_grade = grades["Kate"]
except KeyError:
    print("no grade for Kate!")
```

Podemos verificar se uma chave existe em um dict usando o operador "in"

In [ ]:
```
joel_has_grade = "Joel" in grades # True
print(joel_has_grade)

kate_has_grade = "Kate" in grades # False
print(kate_has_grade)
```

Dicts possuem um método "get" que devolve um valor padrão ao invés de disparar uma exceção caso uma chave não exista

In [ ]:
```
joels_grade = grades.get("Joel", 0) # equals 80
print(joels_grade)

kates_grade = grades.get("Kate", 0) # equals 0
print(kates_grade)
```

Pode-se atribuir valores a um dict usando o mesmo formato de colchetes

In [ ]:
```
print(grades)

grades["Tim"] = 99 # replaces the old value
print(grades)

grades["Kate"] = 100 # adds a third entry
print(grades)

num_students = len(grades) # equals 3
print(num_students)
```

É comum usarmos dict para representar uma estrutura de dados

In [ ]:
```
tweet = {
    "user" : "joelgrus",
    "text" : "Data Science is Awesome",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yol
o"]
}

print(len(tweet))
```

Trabalhando com os itens de um dicionário

In [ ]:
```
tweet_keys = tweet.keys() # list of keys
print(tweet_keys)

tweet_values = tweet.values() # list of values
print(tweet_values)

tweet_items = tweet.items() # list of (key, value) tuples
print(tweet_items)

print("user") in tweet_keys # True, but uses a slow list in
print("user") in tweet # more Pythonic, uses faster dict in
print("joelgrus") in tweet_values # True
```

defaultdict

Imagine que você está tentando contar as palavras em um documento. Uma abordagem óbvia é criar um dicionário no qual as chaves são palavras e os valores são contagens. À medida que você verifica cada palavra, você pode incrementar sua contagem se já estiver no dicionário e adicioná-la ao dicionário se não existir

In [ ]:
```
document =  """This book uses the word in a more restricted sense: hacki
ng is a
               recreational and educational sport. It consists of attemp
ting to make
               unauthorised entry into computers and to explore what is
 there. The
               sport's aims and purposes have been widely misunderstood;
most
               hackers are not interested in perpetrating massive fraud
s, modifying
               their personal banking, taxation and employee records, or
inducing
               one world super-power into inadvertently commencing Armag
eddon in the
               mistaken belief that another super-power is about to atta
ck it. Every
               hacker I have ever come across has been quite clear about
where the
               fun lies: it is in developing an understanding of a syste
m and
               finally producing the skills and tools to defeat it. In t
he vast
               majority of cases, the process of 'getting in' is much mo
re
               satisfying than what is discovered in the protected compu
ter files."""

word_counts = {}
for word in document.split(' '):
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1

print(word_counts)
```

Podemos melhor capturando possíveis exceções:

In [ ]:
```
word_counts = {}
for word in document.split(' '):
    try:
        word_counts[word] += 1
    except KeyError:
        word_counts[word] = 1

print(word_counts)
```

Ficará ainda melhor se usarmos o método GET de dict:

In [ ]:
```
word_counts = {}
for word in document.split(' '):
    previous_count = word_counts.get(word, 0)
    word_counts[word] = previous_count + 1
print(word_counts)
```

Uma melhor opção é usar defaultdict.
Um defaultdict é como um dicionário regular, exceto que, quando você tenta procurar uma chave que não contém ele primeiro adiciona um valor para isso usando uma função de argumento zero que você forneceu ao criá-lo.

In [ ]:
```
from collections import defaultdict

word_counts = defaultdict(int) # int() produces 0
for word in document.split(' '):
    word_counts[word] += 1

print(word_counts)
```

In [ ]:
```
dd_list = defaultdict(list) # list() produces an empty list
print(dd_list)
dd_list[2].append(1) # now dd_list contains {2: [1]}
print(dd_list)

dd_dict = defaultdict(dict) # dict() produces an empty dict
print(dd_dict)
dd_dict["Joel"]["City"] = "Seattle" # { "Joel" : { "City" : Seattle"}}
print(dd_dict)

dd_pair = defaultdict(lambda: [0, 0])
print(dd_pair)
dd_pair[2][1] = 1 # now dd_pair contains {2: [0,1]}
print(dd_pair)
```

Counter
O principal uso de Counter é criar histogramas

In [ ]:
```
from collections import Counter
c = Counter([0, 1, 2, 0]) # c is (basically) { 0 : 2, 1 : 1, 2 : 1 }
print(c)
```

Isso resolve de maneira simples nosso problema de contagem de palavras:

In [ ]:
```
word_counts = Counter(document.split(" "))
print(word_counts)
```

Counter possui um método chamado most_commom que é muito usado

In [ ]:
```
for word, count in word_counts.most_common(30):
    print(word, count)
```

Sets

Representa uma coleção de elementos distintos, ou seja, não repetidos

In [ ]:
```
s = set()
print(s)

s.add(1) # s is now { 1 }
s.add(2) # s is now { 1, 2 }
s.add(2) # s is still { 1, 2 }
print(s)

x = len(s) # equals 2
print(x)

y = 2 in s # equals True
print(y)

z = 3 in s # equals False
print(z)
```

Usaremos SET por duas razões:
- **Primeiro porque o operador IN em Sets é muito rápido, especialmente quanto estamos lidando com grandes quantidades de dados**
- **Segundo porque em alguns casos queremos identificar um grupo distinto em uma coleção**

In [ ]:
```
stopwords_list = ["a","an","at"] + hundreds_of_other_words + ["yet", "yo
u"]
"zip" in stopwords_list # False, but have to check every element

stopwords_set = set(stopwords_list)
"zip" in stopwords_set # very fast to check
```

In [ ]:
```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list) # 6
print(num_items)

item_set = set(item_list) # {1, 2, 3}
num_distinct_items = len(item_set) # 3
print(num_distinct_items)

distinct_item_list = list(item_set) # [1, 2, 3]
```

Controle de Fluxo

O uso do teste condicional IF

In [ ]:
```
if 1 > 2:
    message = "if only 1 were greater than two…"
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

Ternário if-then-else

In [ ]:
```
x = 5
parity = "even" if x % 2 == 0 else "odd"
print(parity)
```

O laço While

In [ ]:
```
x = 0
while x < 10:
    print(x, "is less than 10")
    x += 1
```

Contudo é mais frequente o uso de FOR e IN

In [ ]:
```
for x in range(10):
    print(x, "is less than 10")
```

Para uma lógica mais complexa usamos o CONTINUE e o BREAK

In [ ]:
```
for x in range(10):
    if x == 3:
        continue # go immediately to the next iteration
    if x == 5:
        break # quit the loop entirely
    print(x)
```

Trabalhando com booleans

Em Python, booleans são capitalizados

In [ ]:
```
one_is_less_than_two = 1 < 2 # equals True
print(one_is_less_than_two)

true_equals_false = True == False # equals False
print(true_equals_false)
```

Python usa o valor None para indicar a ausência de valor

In [ ]:
```
x = None
```

In [ ]:
```
print(x) == None # prints True, but is not Pythonic
```

In [ ]:
```
print(x) is None # prints True, and is Pythonic
```

In [ ]:
```
print(x) == None
```

In [ ]:
```

```

Ordenação (sorting)

Toda lista Python possui um método SORT para promover a ordenação do conteúdo na mesma lista. Caso prefira ter uma segunda lista fruto da ordenação, pode-se usar a função SORTED

In [ ]:
```
x = [4,1,2,3]
print(x)
```

In [ ]:
```
y = sorted(x) # is [1,2,3,4], x is unchanged
print(y)
```

In [ ]:
```
x.sort() # now x is [1,2,3,4]
print(x)
```

Por padrão, SORT e SORTED ordenam do menor para o maior, caso queira o inverso, deve-se usar a função SORTED com o parâmetro reverse=True. Você também pode informar uma função que será aplicada a cada elemento e cuja saída será utilizada na ordenação

In [ ]:
```
x = sorted([-4,1,-2,3], reverse=True)
print(x)
```

In [ ]:
```
# sort the list by absolute value from largest to smallest
x = sorted([-4,1,-2,3], key=abs, reverse=True) # is [-4,3,-2,1]
print(x)
```

In [ ]:
```
word_counts.items()
```

In [ ]:
```
# sort the words and counts from highest count to lowest
wc = sorted(word_counts.items(), key=lambda item: item[1], reverse=True
)
print(wc)
```

List Comprehensions

Freqüentemente, você quer transformar uma lista em outra lista, escolhendo apenas determinados elementos, ou transformando elementos, ou ambos. A maneira Pythonic de fazer isso é com List Comprehensions:

In [ ]:
```
even_numbers = [x for x in range(5) if x % 2 == 0] # [0, 2, 4]
print(even_numbers)
```

In [ ]:
```
squares = [x * x for x in range(5)] # [0, 1, 4, 9, 16]
print(squares)
```

In [ ]:
```
even_squares = [x * x for x in even_numbers] # [0, 4, 16]
print(even_squares)
```

Similarmente você pode transformar listas em dicionários ou sets

In [ ]:
```
square_dict = { x : x * x for x in range(5) } # { 0:0, 1:1, 2:4, 3:9, 4:
16 }
print(square_dict)
```

In [ ]:
```
square_set = { x * x for x in [1, -1] } # { 1 }
print(square_set)
```

Se você não precisa do valor da lista, convencionalmente usamos um sublinhado como o variável:

In [ ]:
```
zeroes = [0 for _ in even_numbers] # has the same length as even_numbers
print(zeroes)
```

Uma list comprehension pode incluir múltiplos fors:

In [ ]:
```
pairs = [(x, y)
    for x in range(10)
    for y in range(10)] # 100 pairs (0,0) (0,1) ... (9,8), (9,9)
print(pairs)
```

In [ ]:

In [ ]: