Manual básico de Python

(EM CONSTRUÇÃO)

Última versão atualizada em: 02/04/2019

Por que usar Python?

- Interoperabilidade (comunica-se de forma transparente com outros sistemas ou linguagens)
- Multiplataforma (pode ser utilizada em sistemas Linux, Mac OS ou Windows)
- Robustez
- Velocidade de aprendizado (perfeita para iniciantes em programação)
- Simplicidade
- Livre
- Muitas bibliotecas

Instalação

Você pode instalar a distribuição Anaconda <https://www.anaconda.com/>, que já inclui a linguagem Python e a IDE Spyder, com a maioria das bibliotecas que precisamos para Machine Learning/Data Science inclusas.

Após a instalação do Anaconda. Abra Anaconda Navigator e clique em Launch em Spyder.

Spyder é a IDE que usaremos para desenvolver nossos algoritmos em Python.

O que é uma IDE?

IDE, do inglês *Integrated Development Environment*. O desenvolvimento de um programa requer o uso de várias ferramentas como:

– um "editor de texto" para escrever o programa fonte
– um "interpretador Python" para "rodar" o programa
– um "terminal" onde o programa é "rodado" e permite a entrada e saída dos dados
– um "depurador de programas", que te ajuda a encontrar erros

● O IDE (Spyder) permite que você trabalhe em um só ambiente e não fique trocando (entrando e saindo) de uma ferramenta para outra.

The Zen of Python

O Python tem um tipo de "descrição de princípios básicos para codar", que são recomendações importantes (Pythonic) para que possamos utilizar a linguagem da melhor forma possível. Ela pode ser encontrada dentro do Python simplesmente digitando na linha de comando:

➢ import this

O Zen do Python, por Tim Peters

- ✓ Bonito é melhor que feio.
- ✓ Explícito é melhor que implícito.
- ✓ Simples é melhor que complexo.
- ✓ Complexo é melhor que complicado.
- ✓ Linear é melhor do que aninhado.
- ✓ Esparso é melhor que denso.
- ✓ Legibilidade conta.

- ✓ Casos especiais não são especiais o bastante para quebrar as regras.
- ✓ Ainda que praticidade vença a pureza.
- ✓ Erros nunca devem passar silenciosamente.
- ✓ A menos que sejam explicitamente silenciados.
- ✓ Diante da ambiguidade, recuse a tentação de adivinhar.
- ✓ Deveria haver um — e preferencialmente só um — modo óbvio para fazer algo.
- ✓ Embora esse modo possa não ser óbvio a princípio a menos que você seja holandês.
- ✓ Agora é melhor que nunca.
- ✓ Embora nunca frequentemente seja melhor que *já*.
- ✓ Se a implementação é difícil de explicar, é uma má ideia.
- ✓ Se a implementação é fácil de explicar, pode ser uma boa ideia.
- ✓ *Namespaces* são uma grande ideia — vamos ter mais dessas!

Obs.: Um *namespace* é basicamente um sistema para certificar-se que todos os nomes em um programa são únicos e podem ser usados sem qualquer conflito.

Onde buscar matéria para aprender a desenvolver em Python – GRATUITO

- Curso em Vídeo Python 3. < https://www.cursoemvideo.com/course/curso-python-3/>
- Python for Data Science (7 horas) – Kaggle <https://www.kaggle.com/learn/python?utm_medium=social&utm_source=twitter.com&utm_campaign=new%20python%20course%20announcement>
- Grupo de Python no Brasil <https://python.org.br/>
- Introdução à Ciência da Computação com Python (USP) – Coursera

<https://www.coursera.org/learn/ciencia-computacao-python-conceitos\>

- Aulas Prof. Argenta (UFPR): <https://argenta-ma.github.io/ScientificComputing/>
-

Primeiros comandos em Python

Tipos primitivos

int	Número inteiro	7, -4, 0, 9875
float	Número real	4.5, 0.76, 0.099, -15.233, 7.0
bool	Valores lógicos	True, False
str	Caractere ou ‘string’	‘Olá’, ‘Bem vindo’

Verifica tipo de uma variável ou valor digitado:

type(5)
class int
type(‘ML’)
class string

Operadores aritméticos

+	adição
-	subtração
*	multiplicação

/	divisão
**	potência
//	Divisão inteira
%	resto da divisão

Ordem de precedência das operações aritméticas

1. ( )
2. (**)
3. (*, /, //, %)
4. (+, -)

Ou seja, em uma operação aritmética, o Python primeiro calcula o que estiver dentro de parênteses, depois calcula as potências e depois em sequência o que aparecer primeiro na ordem apresentada. Na dúvida, sempre use parênteses nas operações. E não esqueça de testar seus cálculos.

Operadores relacionais

=	atribuição
==	igualdade
!=	Diferença
>=	maior ou igual
<=	menor ou igual
>	maior
<	menor

Utilizando módulos

Certas características do Python não são carregadas por *default*. Para usá-las, precisamos importar os módulos que as contém.

import **módulo**	Importa todas as funcionalidades do módulo
from **módulo** import **funcionalidade**	Importa funcionalidade que você escolher
from **módulo** import **funcionalidade 1, funcionalidade 2, ...**	Importa várias funcionalidades escolhidas

Exemplos de módulos:
import math

ceil	Arredondamento para cima
floor	Arredondamento para baixo
trunc	Trunca números, da vírgula para frente sem arredondamento
pow	Potência
sqrt	Raiz quadrada
factorial	Fatorial

Ex.:
raiz_quadrada = math.sqrt(2)

Depois de importar um módulo, para acessar qualquer objeto (sqrt: raiz quadrada) você precisa usar math como um prefixo.

Mas se você precisa apenas de um valor específico que está dentro de um módulo, você pode importar da seguinte forma:
from math import sqrt
E assim, usá-lo sem qualquer prefixo.
raiz_quadrada = sqrt(2)
Mas tomar cuidado, porque dessa forma não sabemos de qual módulo é o objeto que estamos chamando, e dado que mais de um módulo pode conter o mesmo nome de objeto, isso pode gerar confusões.
Algumas vezes também usamos uma abreviação na hora de importar nossos módulos, quando estes têm nomes muito grandes, por exemplo, o que pode ser chato se tivermos que digitar isso várias vezes, como é o caso do módulo que plota gráficos.
Ex.:
import matplotlib.pyplot as plt

- plt é uma abreviação usual para este módulo

plt.plot(X)

Condições (Básico)

- Representação estrutura ou identada

```
if condição:
    bloco TRUE
else:
    bloco FALSE
```

- Apenas um dos blocos é executado

Ex.:

```
tempo = int(input('Quantos anos tem o seu carro? '))
if tempo <= 3:
    print('Carro novo')
else:
    print('Carro velho')
print('--FIM—')
```

- Todo comando alinhado à esquerda será executado, o comando identado (alinhado para 'dentro') pode ser executado ou não.

Condições aninhadas
if condição:
elif:
else:
Estrutura de repetição for

- Laços, repetições ou iterações

for c in range(1, 10):
    bloco de ações

Estrutura de repetição while

- Faz uma repetição enquanto uma condição for verdadeira

while ()
if:
elif
elif

Interrompendo repetições while
Comando break

Estruturas de dados

Listas

Provavelmente esta é a estrutura de dados mais fundamental em Python.
Uma lista é uma coleção simples e ordenada de valores (de qualquer tipo, string, int, float, etc)
Listas são objetos para armazenar uma sequência de coisas (objetos).

Até então, uma variável era relacionada a um número (ou cadeia de caracteres), mas algumas vezes temos coleções de números, por exemplo, temperatura em graus celsius: −20,−15,−10,−5, 0,…,40−20,−15,−10,−5,0,…,40.
Solução simples: uma variável para cada valor:

```
C1 = -20

C2 = -15

C3 = -10

...

C13 = 40
```

Solução muito tediosa para muitos valores!
Melhor: um conjunto de valores podem ser colecionados em uma lista:

```
C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
```

Agora temos apenas uma variável, C, armazenando todos os valores.

Exemplos de listas:

lista_inteiro = [1, 2, 3]
list_string = [‘maria’, ‘joao’, ‘felicio’]
list_mix = [‘maria’, 0.1, true]

Podemos ver o tamanho de uma lista através do comando len ou a soma de seus valores através de sum.

list_length = len(lista_inteiro)
#igual a 3
list_sum = sum(lista_inteiro)
#igual a 6

Você pode obter o i-ésimo valor de uma lista com colchetes

n1= C[0] # listas começam com índice 0, esse é o primeiro elemento da lista
um = C [1] #segundo elemento da lista
nove = C [-1] #‘pythonic’ para o último elemento
oito = C [-2] # penúltimo elemento
C [0] = -1 # agora o elemento de índice 0 se torna -1

Podemos também usar colchetes para “slice” listas

primeiros_tres = C [:3]
do_terceiro_ao_ultimo = C [3:]
ultimos_tres = C [-3:]
sem_primeiro_e_ultimo = C [1:-1]

Python tem um operador in para checar componentes de uma lista:

1 in C # True
0 in C # False

Concatenar listas

C.extend([4, 5, 6]) # Adiciona os novos elementos no final da lista
ou
y = C + [4, 5, 6]
Adiciona um valor por vez
C.append(0)

Operações em listas: anexar, estender, inserir, excluir

Através dos comandos append, extend, insert, delete, podemos anexar um elemento a mais ao final da lista, extender a lista com outra lista, inserir um novo elemento em determinada posição e apagar um elemento, respectivamente.

```
>>> C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]

>>> C.append(35)   # adiciona um novo elemento 35 ao final

>>> C

[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]

>>> C = C + [40, 45]     # extende C no final

>>> C

[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

```
>>> C.insert(0, -15)     # insere -15 como índice 0
>>> C
[-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del C[2]             # apaga o 3o elemento
>>> C
[-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del (C[2])       # apaga o que é agora o 3o elemento
>>> C
[-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> len(C)               # comprimento da lista
11
```

Operações em listas: busca por elementos, índices negativos
Podemos buscar um elemento em uma lista utilizando a classe .index() da lista, descobrir se um elemento está na lista com condicionais e buscar o último elemento com índices negativos.

```
>>> C.index(10)   # índice do primeiro elemento com valor 10
3
>>> 10 in C       # 10 é um elemento de C?
True
>>> C[-1]         # o último elemento da lista
45
>>> C[-2]         # o penúltimo elemento da lista
40
>>> umalista = ['book.tex', 'book.log', 'book.pdf']
>>> texfile, logfile, pdf = umalista  # atribuição dos valores direto à variáveis
>>> texfile
'book.tex'
```

```
>>> logfile
'book.log'
>>> pdf
'book.pdf'
```

Tuplas

Tuplas são listas imutáveis, ou seja, que não podem ser alteradas. Para especificar uma tupla, use um parêntese em vez de colchetes.
Ex.:
My_list = [1, 2]
My_tuple = (1, 2)

```
>>> t = (2, 4, 6, 'temp.pdf')    # definindo uma tupla
>>> t =  2, 4, 6, 'temp.pdf'     # podemos fazer sem o parênteses
>>> t[1] = -1
...
TypeError: object does not support item assignment


>>> t.append(0)
...
AttributeError: 'tuple' object has no attribute 'append'


>>> del t[1]
...
TypeError: object doesn't support item deletion
```

Tuplas podem fazer muito de que listas podem:

```
>>> t = t + (-1.0, -2.0)          # adicionando duas tuplas
```

```
>>> t
(2, 4, 6, 'temp.pdf', -1.0, -2.0)
>>> t[1]                    # indexando
4
>>> t[2:]                   # subtupla/fatia
(6, 'temp.pdf', -1.0, -2.0)
>>> 6 in t                  # elemento está na tupla?
True
```

Porque usar tuplas se as listas tem mais funcionalidades?
Tuplas são constantes e por isso protegidas contra mudanças acidentais;
Tuplas são mais rápidas que as listas;
Tuplas poder ser utilizadas como chaves em dicionários (listas não, veremos isso mais adiante...).
Funcionalidades de listas

Construção	**Significado**
`a = []`	inicializa uma lista vazia
`a = [1, 4.4, 'run.py']`	inicializa a lista
`a.append(elem)`	adiciona o objeto `elem` ao final
`a + [1,3]`	adiciona duas listas
`a.insert(i, e)`	insere o elemento `e` no índice `i`
`a[3]`	elemento da lista na posição 3
`a[-1]`	pega o último elemento da lista

Construção	Significado
`a[1:3]`	corte: copia dos dados para uma sublist (aqui: índices 1, 2)
`del a[3]`	apaga um elemento (índice `3`)
`a.remove(e)`	remove um elemento com o valor `e`
`a.index('run.py')`	busca o índice correspondente ao valor do elemento
`'run.py' in a`	testa se um valor está na lista
`a.count(v)`	conta quantos elementos possuem o valor `v`
`len(a)`	número dos elementos na lista `a`
`min(a)`	o menor elemento na lista `a`
`max(a)`	o maior elemento na lista `a`
`sum(a)`	soma todos os elementos de `a`
`sorted(a)`	retorna uma versão ordenada da lista `a`
`isinstance(a, list)`	é `True` se `a` é uma lista
`type(a) is list`	é `True` se `a` é uma lista

Dicionários

Esta estrutura de dados associa valores a ‘keys’ e permite acessar rapidamente um valor correspondente a uma key.

```
nota = {‘Joel’: 80, ‘Tim’: 95
}
```

Você pode procurar por um valor chave usando colchetes

```
Joels_nota =  nota[‘Joel’]
#equals 80
```

Funções

Uma função é uma regra para pegar uma ou mais entradas e retornar uma saída correspondente.

Em Python, definimos funções usando def

```
def double(x):
    ‘’’aqui em geral é colocado uma ‘docstring’ opcional, que explica o que a função faz por exemplo, esta função multiplica a entrada por 2’’’
    return x * 2
```

Outro exemplo:

```
def dobra(lst):
   '''essa função dobra cada valor de uma lista'''
   i = 0
   while i < len(lst):
      lst[i] *= 2
      i += 1
```

REFERÊNCIAS

Aulas de Introdução à Computação Científica (Estruturas – UFPR). Disponível em: https://argenta-ma.github.io/ScientificComputing/. Acesso em: 02/04/2019