



## **Fundamentos do Bootcamp**

**Bootcamp IGTI: Desenvolvedor Python**

**Túlio Philipe Ferreira e Veiria**

**2020**

## **Fundamentos do Bootcamp**

Bootcamp IGTI: Desenvolvedor Python

Túlio Philipe Ferreira e Veiria

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

## Sumário

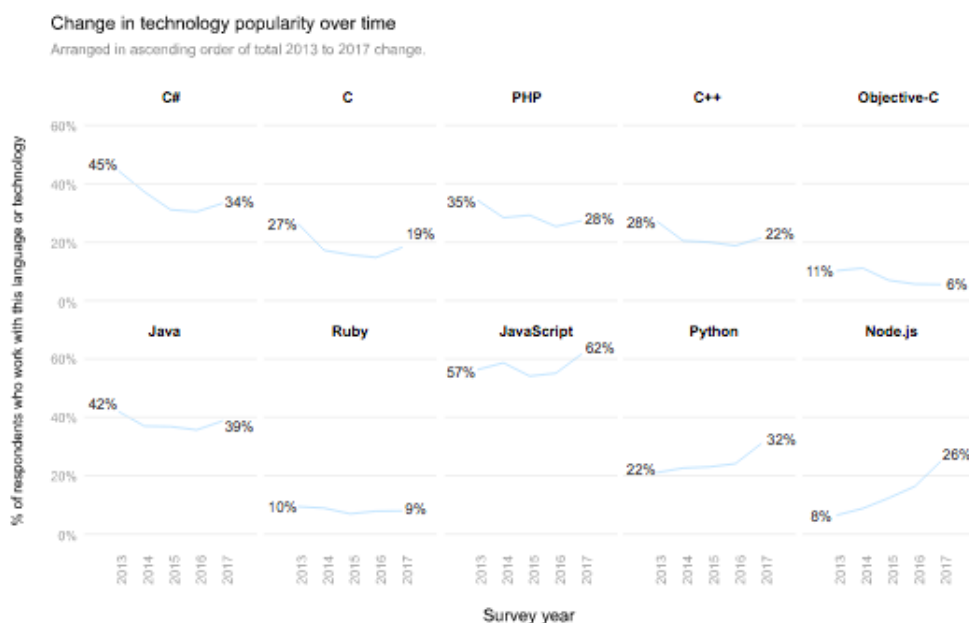
---

Capítulo 1. Introdução ao Python .....	4
Instalação e configuração do ambiente .....	6
Segunda forma de utilização dos códigos .....	9
Capítulo 2. Estrutura de Desenvolvimento .....	11
Estrutura e sintaxe do Python .....	11
Estrutura e tipos de dados .....	13
Capítulo 3. Dados em Python .....	15
Tipos de dados em Python .....	15
Dados numéricos .....	15
Strings .....	17
Listas .....	17
Conjuntos (sets) .....	20
Dicionários (dict) .....	21
Tuplas .....	22
Capítulo 4. Controle de Fluxo .....	23
Condicionais .....	23
Construção de Loops .....	24
Funções .....	26
Classes .....	26
Referências .....	28

## Capítulo 1. Introdução ao Python

Python é uma linguagem de programação concebida na década de 1980, desenvolvida por Guido van Rossum no Centrum Wiskunde and Informatica (CWI) na Holanda. Python foi proposta para ser a sucessora da linguagem ABC. Segundo dados da TIOBE (2019), a linguagem Python continua a ser a linguagem de programação que mais cresce no mundo. Segundo projeções, dentro de 3 a 4 anos o Python deve ser a linguagem de programação mais utilizada no mundo, superando a JAVA e C. A Figura 1 apresenta a variação de popularidade das diferentes linguagens de programação durante os anos.

**Figura 1 – Variação da popularidade das linguagens de programação.**



Fonte: OVERFLOW, 2017.

Algumas características que alavancam o crescimento dessa linguagem, são:

- **Facilidade de aprendizado.**

Como essa linguagem apresenta uma programação em alto nível e que se assemelha com a língua inglesa escrita, implementar algoritmos torna-se uma tarefa menos custosa.

- **Linguagem portátil e expansividade.**

Com o Python é possível integrar componentes do JAVA e dotNet, além de possibilitar a integração com bibliotecas do C e C++. O Python também é suportado pela maioria das plataformas existentes, por exemplo, MAC, Windows, Linux etc.

- **Escalabilidade.**

Utilizando a linguagem Python é possível criar pequenos programas para prototipagem ou desenvolver grandes sistemas. Existem várias bibliotecas para o desenvolvimento de aplicações.

- **Grande comunidade.**

Python é utilizada por grandes empresas e por diferentes centros de pesquisas. Engenheiros de software, matemáticos, cientistas de dados, biólogos e, praticamente, todas as áreas da ciência moderna, utilizam das funcionalidades da linguagem Python para desenvolver projetos. Assim, existe uma extensa comunidade com conhecimento em várias áreas que auxiliam na construção de bibliotecas e na solução de problemas.

Python foi desenvolvida para ser uma linguagem interpretada e de script. Entretanto, utilizando os conceitos de programação orientada a objetos e os construtores, a linguagem Python pode ser utilizada como qualquer outra linguagem naturalmente orientada a objeto. Funcionalidades como classes, objetos e métodos, além dos princípios de abstração, como encapsulamento, herança e polimorfismo, estão presentes nessa linguagem. Assim, utilizar os conceitos de orientação a objeto para desenvolver aplicações é algo similar a outras linguagens.

O desenvolvimento da linguagem Python foi realizado utilizando-se os preceitos de que um código simples e fluido é mais elegante e fácil de utilizar que códigos difíceis de interpretar e que empregam otimizações prematuras. Portanto, Python apresenta-se como uma linguagem fácil de interpretar, com alto nível de abstração, em que a experiência do usuário é o foco da aplicação.

Como mostrado anteriormente, Python é uma linguagem que preza pela simplicidade e elegância da programação. Para se ter uma ideia desses conceitos, você pode acessar os 20 princípios de desenvolvimento dessa linguagem, diretamente, pelas linhas de comando. Assim, você verá os preceitos do Zen of Python.

Existem duas principais versões do Python: 2.x e 3.x. O x significa a atualização da versão. Essas duas versões apresentam praticamente as mesmas funcionalidades, entretanto existe incompatibilidade entre algumas funções. O exemplo mais comum de incompatibilidade é o print. No Python 2.x ela é um statement, já no Python 3 foi transformada em função. Durante todo o nosso curso nós vamos utilizar o Python 3.

A seguir são apresentadas duas opções de utilização do Python 3. A primeira corresponde em instalar em sua própria máquina todos os pacotes necessários. A segunda consiste em utilizar uma conta do Google e a plataforma Google Colaboratory. Utilizando qualquer uma das opções, ou qualquer outra *Integrated Development Environment* (IDE), você estará apto a implementar todos os exemplos de aplicações apresentados nesta disciplina.

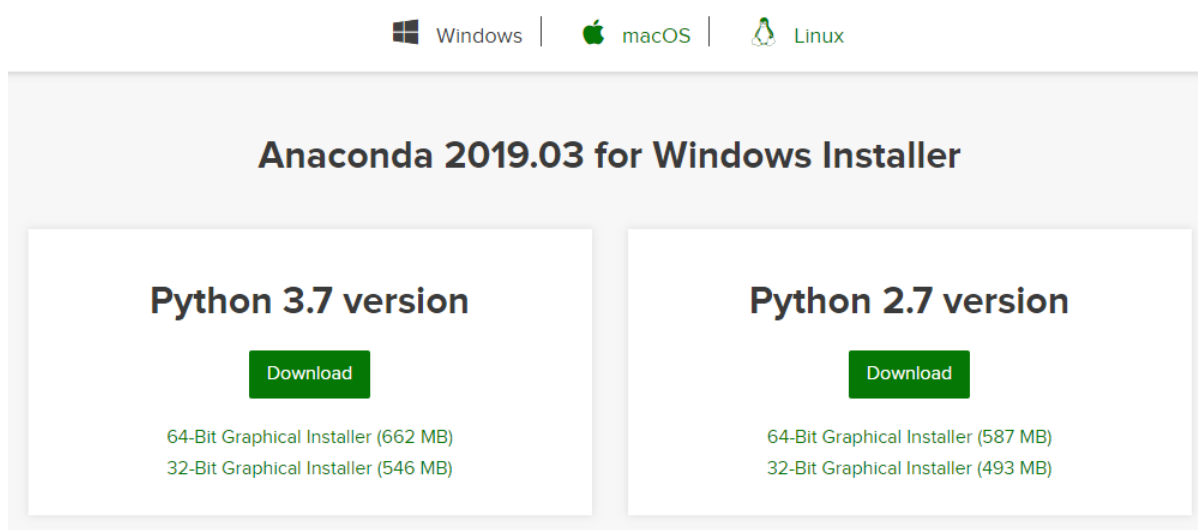
### Instalação e configuração do ambiente

Existem vários ambientes integrados de desenvolvimento (IDE) para Python. Sugiro utilizar a distribuição Python Anaconda que já possui a IDE Spyder para desenvolvimento. Anaconda é uma distribuição desenvolvida especialmente para aplicações científicas. Com a instalação do Anaconda, vários pacotes como o scikit learn e numpy já vem instalados e configurados. Assim, o seu computador já está, praticamente, configurado para a utilização de todos os códigos desta disciplina.

Para a instalação do Anaconda, siga os passos a seguir:

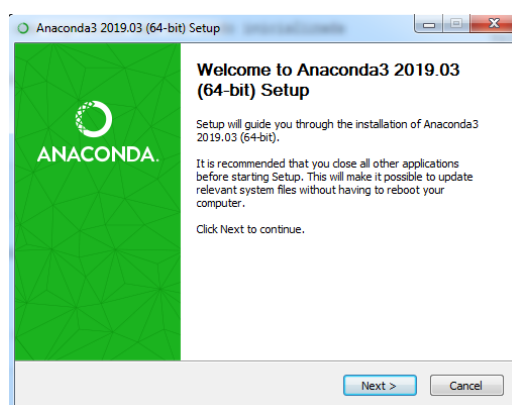
1. Baixe e instale o Anaconda (Windows) pelo link: <https://www.anaconda.com/distribution/>.

**Figura 2 – Tela de Download do Anaconda.**



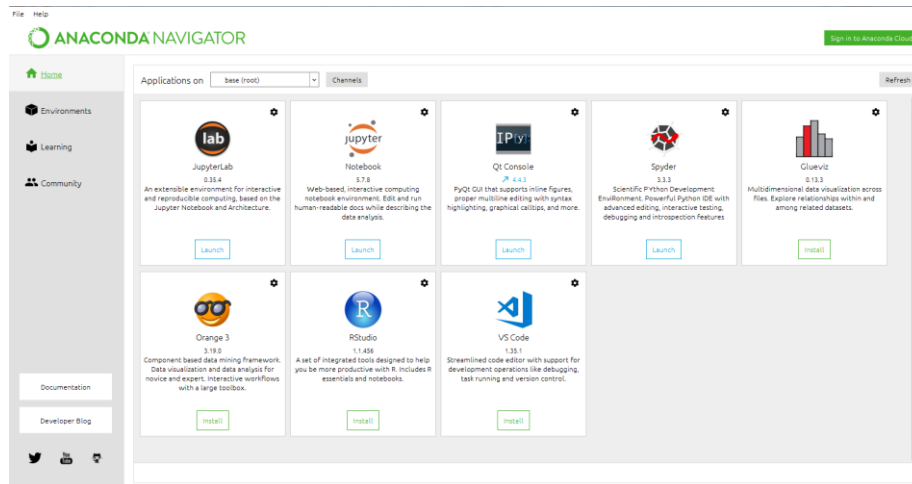
2. Escolha a versão Python 3.x.
3. Será baixado para a sua máquina o instalador do Anaconda com, aproximadamente, 700MB.
4. Inicie o instalador do Anaconda.

**Figura 3 – Tela inicial de instalação do Anaconda.**



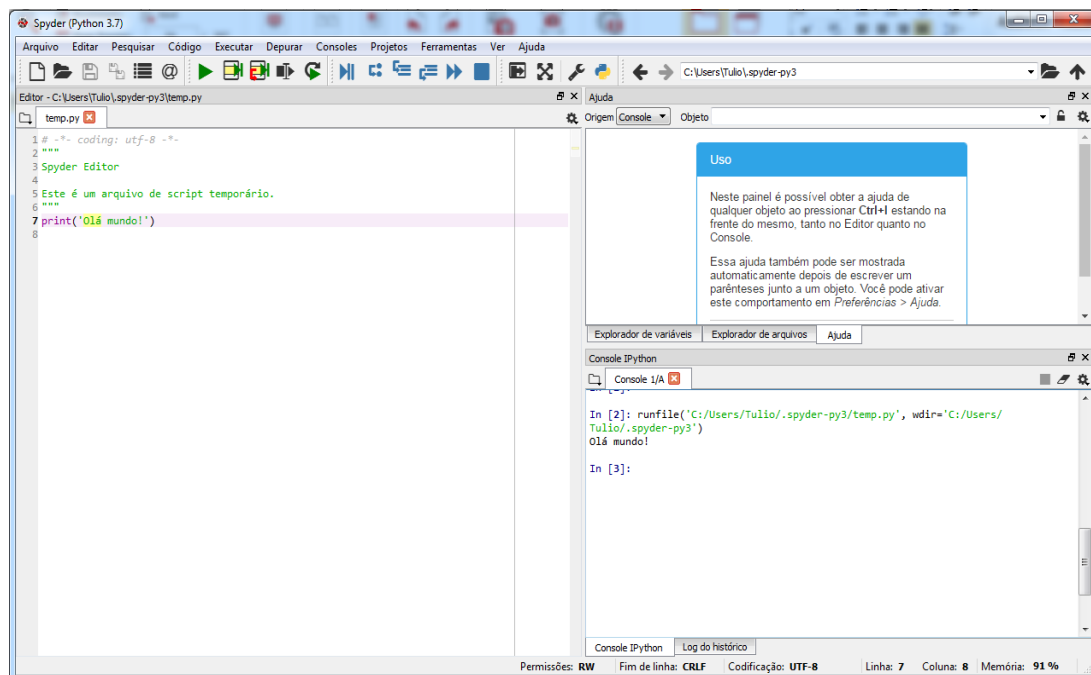
5. Recomendo utilizar a instalação padrão da distribuição.
6. Após a instalação, você verá uma tela igual a esta:

**Figura 4 – Tela inicial do Anaconda.**



7. Para iniciar a IDE Spyder clique no ícone correspondente, presente na Figura 4.
8. Pronto, você já pode executar o `print('Olá mundo!')`.

**Figura 5 – Olá mundo em Python.**



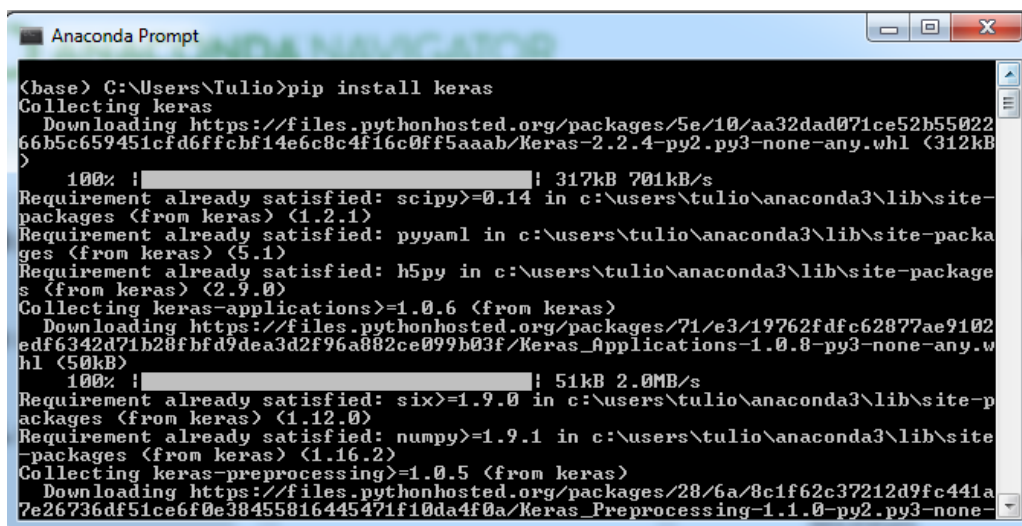
9. Para instalar outros pacotes Python no Anaconda, vá ao terminal (prompt de comando) do Anaconda e utilize o comando:



*pip install nomedopacote*

Por exemplo, a instalação do pacote Keras.

**Figura 6 – Prompt de comando do Anaconda.**



```

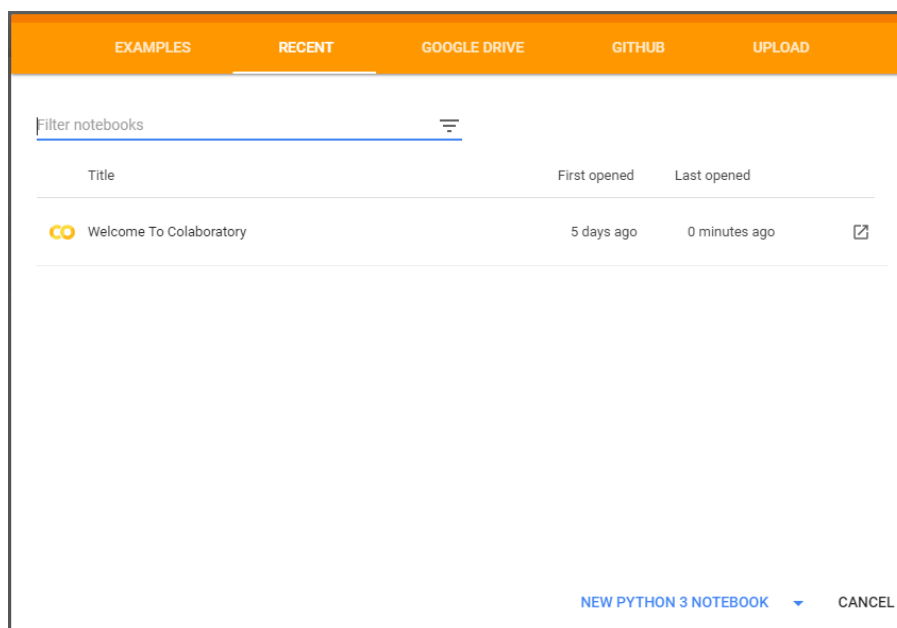
<base> C:\Users\tulio>pip install keras
Collecting keras
  Downloading https://files.pythonhosted.org/packages/5e/10/aa32dad071ce52b5502266b5c659451cfd6ffcbf14e6c8c4f16c0ff5aaab/Keras-2.2.4-py2.py3-none-any.whl (312kB)
    100% |#####| 317kB 701kB/s
Requirement already satisfied: scipy>=0.14 in c:\users\tulio\anaconda3\lib\site-packages (from keras) (1.2.1)
Requirement already satisfied: pyyaml in c:\users\tulio\anaconda3\lib\site-packages (from keras) (5.1)
Requirement already satisfied: h5py in c:\users\tulio\anaconda3\lib\site-packages (from keras) (2.9.0)
Collecting keras-applications=1.0.6 (from keras)
  Downloading https://files.pythonhosted.org/packages/71/e3/19762fd6c62877ae9102edf6342d71b28fbfd9dea3d2f96a882ce099b03f/Keras_applications-1.0.8-py3-none-any.whl (50kB)
    100% |#####| 51kB 2.0MB/s
Requirement already satisfied: six>=1.9.0 in c:\users\tulio\anaconda3\lib\site-packages (from keras) (1.12.0)
Requirement already satisfied: numpy>=1.9.1 in c:\users\tulio\anaconda3\lib\site-packages (from keras) (1.16.2)
Collecting keras-preprocessing=1.0.5 (from keras)
  Downloading https://files.pythonhosted.org/packages/28/6a/8c1f62c37212d9fc441a7e26736df51ce6f0e38455816445471f10da4f0a/Keras_Preprocessing-1.1.0-py2.py3-none-

```

## Segunda forma de utilização dos códigos

Caso você não deseje instalar nenhum componente em sua máquina, uma opção é utilizar o **Google Colaboratory**. Para acessar o Google Colab é necessário apenas ter uma conta Google e acessar diretamente em seu *browser*. A Figura 7 apresenta a tela inicial do Google Colab.

**Figura 7 – Tela inicial do Google Colaboratory.**



Para a utilização desse ambiente não é necessário instalar nenhum programa adicional. Entretanto, para realizar o upload de algum dataset, por exemplo, são necessários comandos extras. Todos os comandos necessários para realizar o upload dos dataset estão presentes nos códigos disponibilizados. Para iniciar um novo colab, basta clicar em “*NEW PYTHON 3 NOTEBOOK*” e começar a desenvolver a sua aplicação utilizando Python 3.x.

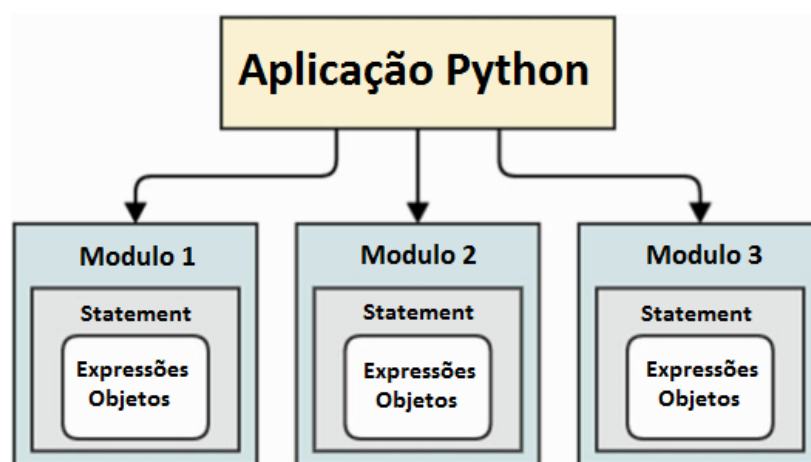
## Capítulo 2. Estrutura de Desenvolvimento

Neste capítulo serão apresentados os principais elementos para a construção de aplicações que utilizem o Python. São apresentadas as “palavras-chave” e como podemos utilizar a “sintaxe” do Python para construir os nossos programas.

### Estrutura e sintaxe do Python

Para a construção de qualquer programa em qualquer linguagem de programação, é necessário conhecer a estrutura que governa o desenvolvimento e as hierarquias de construção. Em Python, podemos dizer que **tudo é um objeto**, funções, estrutura de dados, classes e tipos são objetos. A Figura 8 apresenta um diagrama que representa essa hierarquia.

**Figura 8 – Hierarquia de um programa em Python.**



Fonte: adaptado de SARKAR (2016).

Como dito anteriormente, a estrutura básica em Python são os objetos. Todas as operações são realizadas por meio da manipulação de objetos. Em Python, assim como outras linguagens de programação, possui palavras reservadas. A lista com algumas dessas palavras e descrição são apresentadas na Tabela 1.

**Tabela 1 – Exemplo de algumas palavras-chave e descrição.**

Palavra Chave	Descrição	Exemplo de utilização
and	Operador lógico AND	(5==5 and 1==2) == False
as	Utilizado como sinônimo de algum objeto/referência	import matplotlib.pyplot as plt
class	Utilizada para criar uma classe (POO)	class GeraNumerosAleatorios()
def	Define funções	def adicao(a,b)
del	Deleta referências	del erro
elif	Condicional Else-if (então-se)	If num==1: print('1') elif num==2: print('2')
else	Condicional Else (então)	If num==1: print('1') else: print('nao e 1')
False	Booleano falso	False==0
for	Loop for	for num in array: print('num')
from	Importa componentes específicos dos módulos	from sklearn import metrics
if	Condicional if	if num==1: print('1')
import	Importa um modulo existente	import numpy

in	Procura ou realiza um loop sobre algum objeto existente	for num in array \ if x in y
is	Utilizado para checar igualdade	type('string') is str
not	Operador lógico not	not 1==2
or	Operador lógico or	1 or 2 ==1
return	Retorna objeto(s) de uma função existente	return a,b
while	O loop while	while True: print('valor')

Fonte: adaptado de SARKAR (2016).

### Estrutura e tipos de dados

Como dito anteriormente, todos os elementos em Python são objetos. Cada objeto possui propriedades específicas que são utilizadas para distingui-los. Essas propriedades são:

- Identidade: é única. É criada no momento em que o objeto é criado. Normalmente é representado pelo endereço de memória do objeto.
- Tipo (type): determina o tipo de objeto.
- Valor (value): representa o valor real (atual) armazenado pelo objeto.

A Figura 9 apresenta as propriedades de um objeto *string* que foi criado, e através dela é possível perceber a ação dessas propriedades na prática.

**Figura 9 – Apresentação das propriedades de um objeto.**

```
[3] minha_string = "Disciplina Aplicações em Deep Learning" #armazena a string
```

```
[4] id(minha_string) #identificador do objeto
```

```
↳ 139908227640056
```

```
[5] type(minha_string) #tipo de objeto
```

```
↳ str
```

```
[6] minha_string #mostra a string armazenada
```

```
↳ 'Disciplina Aplicações em Deep Learning'
```

## Capítulo 3. Dados em Python

Neste capítulo são apresentados os principais tipos de dados existentes em Python e como eles são utilizados no desenvolvimento de aplicações. A combinação desses diferentes tipos de dados constitui um dos principais elementos que tornam esta linguagem bastante atrativa.

### Tipos de dados em Python

Em Python existem vários tipos de dados. Nesta apostila vamos falar apenas dos mais importantes para o desenvolvimento de nossas aplicações e, quando existir a necessidade, as estruturas que não apresentadas neste tópico serão discutidas no momento da apresentação dos conceitos.

### Dados numéricos

Existem, basicamente, três tipos de dados numéricos em Python. Inteiros (*integers*), pontos flutuantes (*floats*) e números complexos. A Figura 10 apresenta algumas operações básicas com números inteiros.

**Figura 10 – Algumas operações básicas com números inteiros.**

```
[ ] numero_inteiro= 1234 # representação de numero inteiro
[ ] type(numero_inteiro)
↳ int
[ ] numero_inteiro + 10 # adição
↳ 1244
[ ] numero_inteiro - 10 #subtração
↳ 1224
[ ] numero_inteiro * 2 # multiplicação
↳ 2468
[ ] numero_inteiro / 2 # divisão
↳ 617.0
```

Para os números em pontos flutuantes, podemos representá-los por meio de valores decimais ou em notação de expoente (e ou E seguido de +/- indicando o sentido da notação). A Figura 11 apresenta alguns exemplos de operações utilizando pontos flutuantes.

**Figura 11 – Algumas operações básicas com ponto flutuante.**

```
[19] 1.5 + 3.7
↳ 5.2

[20] 12e3 + 5e3
↳ 17000.0

[21] 2.5e-3
↳ 0.0025

[22] 2.5e-3 + 0.0025
↳ 0.005
```

Os números complexos são representados por dois componentes distintos. Eles possuem uma parte real representado por um *float*, e uma parte imaginária, representada por um *float* seguido da letra *j*. A Figura 12 apresenta algumas operações realizadas com números complexos.

**Figura 12 – Algumas operações básicas com números complexos.**

```
[24] numero_complexo = 2 + 5j # representação de um número complexo

[25] numero_complexo + 3
↳ (5+5j)

[26] numero_complexo + 6j
↳ (2+11j)

[30] type(numero_complexo)
↳ complex

[28] numero_complexo.imag
↳ 5.0

[29] numero_complexo.real
↳ 2.0
```



## Strings

As *strings* são sequências de caracteres utilizadas para representar texto. As *strings* podem armazenar dados textuais ou bytes como informação. Como dito anteriormente, as *strings* são objetos que possuem vários métodos já implementados que permitem manipulá-las de maneira mais conveniente. Uma característica das *strings* é que elas representam objetos imutáveis, ou seja, toda operação realizada com uma *string* cria uma nova *string*. A Figura 13 apresenta algumas operações realizadas com *strings*. Como pode ser visto, existem várias operações que podem ser realizadas sobre os objetos *string*.

**Figura 13 – Algumas operações com strings.**

```
[2] s1 = "Esta é a disciplina"

[3] s2 = " Aplicações em Deep Learning"

[4] print(s1,s2)

↳ Esta é a disciplina Aplicações em Deep Learning

[5] print(s1+'\n'+s2)

↳ Esta é a disciplina
  Aplicações em Deep Learning

[6] ''.join([s1,s2]) #concatena a lista de strings

↳ 'Esta é a disciplina Aplicações em Deep Learning'

[7] s1[::-1] # percorre a string de maneira inversa

↳ 'anilpicsid a é atsE'
```

## Listas

Em Python, listas podem ser um conjunto de objetos heterogêneos ou homogêneos. Nas listas, os objetos são ordenados seguindo a sequência em que foram adicionados na lista. Cada um dos elementos na lista possui o seu próprio índice. Cada objeto pode ser acessado através da referência ao índice que possui. A

Figura 14 apresenta alguns exemplos de como as listas podem ser criadas. Na Figura 15 são mostradas algumas operações sobre listas.

**Figura 14 – Exemplos de criação e acesso às listas.**

```
[8] l1= ['azul','branco', 'amarelo', 'vermelho'] #exemplo de criação de listas
[9] l2= list ([1,'coxinha',10,'pasteis',0.25e-3]) #exemplo de criação de listas
[10] l3= [1,2,3,['a','b','c'], ['IGTI','ADL']]
[12] print(l1,l2)
↳ ['azul', 'branco', 'amarelo', 'vermelho'] [1, 'coxinha', 10, 'pasteis', 0.00025]
[13] print(l3)
↳ [1, 2, 3, ['a', 'b', 'c'], ['IGTI', 'ADL']]
[14] #acessando itens das listas
    l1
↳ ['azul', 'branco', 'amarelo', 'vermelho']
[15] l1[0]
↳ 'azul'
[16] l1[0]+' '+l1[2]
↳ 'azul amarelo'
[17] #acessando intervalos de itens
    l2[1:3]
↳ ['coxinha', 10]
```

**Figura 15 – Exemplos de listas e operações que podem ser executadas.**

```
[28] numeros= list(range(10)) # gera uma lista contendo os numeros de 0 a 9
      numeros[:]
```

```
↳ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[29] numeros[2:5]
```

```
↳ [2, 3, 4]
```

```
[30] #concatenando listas
      numeros*2
```

```
↳ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[32] numeros+12
```

```
↳ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 'coxinha', 10, 'pasteis', 0.00025]
```

```
[33] #acessando listas aninhadas
      13
```

```
↳ [1, 2, 3, ['a', 'b', 'c'], ['IGTI', 'ADL']]
```

```
[34] 13[3]
```

```
↳ ['a', 'b', 'c']
```

```
[35] 13[4]
```

```
↳ ['IGTI', 'ADL']
```

```
[36] 13[3][1]
```

```
↳ 'b'
```

```
[37] #adicionando elemetos à lista
      13.append('Aplicações em Deep Learning')
```

```
[38] 13
```

```
↳ [1, 2, 3, ['a', 'b', 'c'], ['IGTI', 'ADL'], 'Aplicações em Deep Learning']
```

```
[39] #retirando um elemento da lista
      13.pop(3)
```

```
↳ ['a', 'b', 'c']
```

```
13
```

```
↳ [1, 2, 3, ['IGTI', 'ADL'], 'Aplicações em Deep Learning']
```

## Conjuntos (sets)

Os conjuntos ou “sets” são um conjunto de objetos únicos e imutáveis. Para criar esse conjunto é utilizado o método `set( )` ou `{ }`. A Figura 16 apresenta exemplos de criação e operações com esses conjuntos.

**Figura 16 – Exemplos de conjuntos e operações que podem ser executadas.**

```
[41] l1 = [1,2,3,3,3,4,5,5,5,6,6,7,7,7,8,8]. #cria uma lista com numeros repetidos

[42] set(l1)      #cria um conjunto através do método set().

↳ {1, 2, 3, 4, 5, 6, 7, 8}

[43] s1 = set(l1).

[44] #verifica se um elemento pertence ao conjunto
1 in s1

↳ True

[45] 10 in s1

↳ False

[49] s2 = {3,5,10,11,12}    #cria um conjunto através do { }

[50] #operações com conjuntos
s1 - s2    # diferença entre conjuntos

↳ {1, 2, 4, 6, 7, 8}

[51] s1 | s2    #união de conjuntos

↳ {1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12}

[52] s1 & s2    #interseção de conjuntos

↳ {3, 5}

▶ s1 ^ s2    # elementos que nao aparecem, ao mesmo tempo, em ambos

↳ {1, 2, 4, 6, 7, 8, 10, 11, 12}
```

## Dicionários (dict)

Dicionários são conjuntos em Python mutáveis e não ordenados. Eles possuem acesso através de um valor chave 'key' e as chaves são sempre estruturas imutáveis. Eles são criados através do método `dict()`. A estrutura dos dicionários é similar a um JSON. A Figura 17 mostra alguns exemplos de criação e operações com dicionários.

**Figura 17 – Exemplos de dicionários e operações que podem ser executadas.**

```
[55] d1 = {'azul': 3, 'branco': 4, 'verde': 6, 'preto': 1}
d1
↳ {'azul': 3, 'branco': 4, 'preto': 1, 'verde': 6}
```

```
[56] #recuperando itens de um dicionário
d1.get('azul')
↳ 3
```

```
[57] d1['azul']
↳ 3
```

```
[59] # adicionando valores
d1['amarelo']=4
d1
↳ {'amarelo': 4, 'azul': 3, 'branco': 4, 'preto': 1, 'verde': 6}
```

```
[60] d1.keys() #recuperando as chaves
↳ dict_keys(['azul', 'branco', 'verde', 'preto', 'amarelo'])
```

```
[61] d1.values() # recuperando os valores
↳ dict_values([3, 4, 6, 1, 4])
```

```
[65] #cria dicionário utilizando o método dict()
d2 = dict({'laranja': 47, 'roxo': 21, 'azul': 58})
d2
↳ {'azul': 58, 'laranja': 47, 'roxo': 21}
```

```
▶ #atualizando o dicionário d1 utilizando o d2
d1.update(d2)
d1
↳ {'amarelo': 4,
    'azul': 58,
    'branco': 4,
    'laranja': 47,
    'preto': 1,
    'roxo': 21,
    'verde': 6}
```

## Tuplas

Tuplas, assim como as listas, também são sequências de objetos. Entretanto, as tuplas são imutáveis. São utilizadas, normalmente, para representar uma coleção de objetos ou valores. A Figura 18 apresenta as formas de criação e operações que podem ser realizadas com as tuplas.

**Figura 18 – Exemplos de tuplas e operações que podem ser executadas.**

```
[67] #criando tuplas
      t1= (1,)
      t1
```

```
↳ (1,)
```

```
[68] #identificação da tupla
      id(t1)
```

```
↳ 140097858444088
```

```
[69] # modificar o conteúdo da tupla, modifica o endereço
      t1 = t1 + (2,3,4,5,6)
      t1
```

```
↳ (1, 2, 3, 4, 5, 6)
```

```
[70] id(t1)
```

```
↳ 140097840249928
```

```
[71] # tuplas são imutáveis
      t1[2]= 1000
```

```
↳ -----
      TypeError                                Traceback (most recent call last)
      <ipython-input-71-a5f2253b0903> in <module>()
      ----> 1 t1[2]= 1000

      TypeError: 'tuple' object does not support item assignment
```

SEARCH STACK OVERFLOW

```
[72] tupla = (['IGTI','Disciplina'], ['Aplicações','Deep Learning'])
      tupla[0]
```

```
↳ ['IGTI', 'Disciplina']
```

```
[74] l1,l2 = tupla
      print(l1,l2)
```

```
↳ ['IGTI', 'Disciplina'] ['Aplicações', 'Deep Learning']
```

Python proporciona várias formas de criar controles de fluxo para os programas. Alguns exemplos de controle de fluxo disponíveis são:

- Condicionais If-elif-else.
- Loop for.
- Loop while;.
- Loop break, continue e else.
- Try-except.

A estrutura condicional possibilita a escolha de um grupo de ações e estruturas a serem executadas quando determinadas condições são ou não satisfeitas. Em Python, essas estruturas possuem o seguinte formato:

```
<bloco 1 de códigos>           #somente é executado se a  
                                #condição 1 for verdadeira.
```

```
<bloco 2 de códigos>      # é executado se a condição 1 é
                             #falsa e a 2 for verdadeira
```

```
<bloco 3 de códigos>      #executado apenas quando 1 e 2 forem falsas
```

A Figura 19 mostra alguns exemplos de utilização desses condicionais para comandar o fluxo de ações dentro de um código.

**Figura 19 – Exemplos de construção de condicionais.**

```
[76] variavel = 'azul'
      if variavel == 'azul':
          print("Variável é AZUL")
```

↳ Variável é AZUL

```
[78] variavel = 'branco'
      if variavel == 'azul':
          print("Variável é AZUL")
      elif variavel == 'branco':
          print('Variável é BRANCO')
```

↳ Variável é BRANCO

```
[79] variavel = 'verde'
      if variavel == 'azul':
          print("Variável é AZUL")
      elif variavel == 'branco':
          print('Variável é BRANCO')
      else:
          print('Variável é VERDE')
```

↳ Variável é VERDE

## Construção de Loops

Assim como em outras linguagens, as duas principais estruturas de repetição em Python são: **for** e **while**. Essas estruturas são utilizadas para que a execução de um bloco de comandos seja executada repetidas vezes. Em Python, existe a opção de colocar o comando **else** ao final da execução do loop, assim, depois de finalizada a execução da estrutura de repetição (sem a utilização do comando **break**), os comandos que estiverem dentro do **else** serão executados.

### #Loop for

**for** item **in** lista:      #realiza o loop para cada elemento na lista

<bloco de código>    #executado repetidamente



**else:** #comando opcional, executado apenas se o loop terminar normalmente (sem a utilização do **break**)

#Loop **while**

**while** <condicional>: #repete até que a condição seja satisfeita

< bloco de código> #executado repetidamente

**else:** #comando opcional, executado apenas se o loop terminar normalmente (sem a utilização do **break**)

A Figura 20 exemplifica a utilização dos comandos de repetição **for** e **while**.

**Figura 20 – Exemplos de construção de Loop for e while.**

```
[80] #ilustrando loops
sequencia = range(0,5)
for num in sequencia:
    print(num)

0
1
2
3
4

[82] sum = 0
for num in sequencia:
    sum+= num
    print(sum)

10

[83] sequencia = range(0,5)
for num in sequencia:
    print(num)
else:
    print('loop terminou normalmente')

0
1
2
3
4
loop terminou normalmente

[84] sequencia = range(0,5)
for num in sequencia:
    if num <3:
        print(num)
    else:
        break
else:
    print('loop terminou normalmente')

0
1
2

[85] #ilustrando loop while
num = 5
while num >0:
    print(num)
    num -=1 #necessário para evitar o loop infinito

5
4
3
2
1
```

## Funções

As funções podem ser definidas como um bloco de códigos que será executado apenas quando for invocado. Todas as funções precisam ter uma assinatura def e um nome específico para serem invocadas. Dentro das funções existem códigos que devem ser executados após ela ser invocada. O pseudocódigo a seguir mostra como o bloco funcional é criado.

```
def exemploFuncao (parâmetros): #parâmetros = valores de entrada
```

```
< bloco de códigos>
```

```
return valores                #bloco opcional para retorno da função
```

A Figura 21 mostra a implementação de algumas funções simples.

**Figura 21 – Exemplos de construção de funções em Python.**

```
[87] #exemplo de uma função que retorna o quadrado de um número
def retornaQuadrado(numero):
    return numero*numero

numero=5
print("O quadrado de {} é: {}".format(numero,retornaQuadrado(numero)))

O quadrado de 5 é: 25
```

## Classes

Classes em Python são estruturas que permitem utilizar o paradigma de Orientação a Objetos para desenvolver programas que utilizem os conceitos de objetos, encapsulamento, métodos, herança e polimorfismo. Abaixo é apresentado o pseudocódigo para construção de classes em Python.

```
class NomeDaClasse (ClasseBase):
```

```
    variáveis_da_classe    #variáveis compartilhadas com todas as instâncias
```

```
    def __init__ (self, ...): #construtor da classe
```

#instancia as variáveis que são únicas para cada objeto/instancia

```
self.variaveis_unicas = ...
```

```
def __str__(self): representação em string do objeto/instancia
```

```
    return representação(self)
```

```
def métodos(self, ...):    #criação de métodos da classe
```

```
    <bloco de código>
```

O pseudocódigo mostrado anteriormente indica que a classe NomeDaClasse herda os parâmetros definidos pela classe ClasseBase. O parâmetro self é, normalmente, utilizado como o primeiro parâmetro de cada método, pois é ele o responsável por referenciar a instância ou objeto da classe NomeDaClasse e chamar os métodos correspondentes.

Todos os comandos apresentados nesta apostila podem ser baixados e executados através do seguinte link:

- [https://colab.research.google.com/drive/1flsg5PtC8Wlpyq5ynQfO69cGEX\\_LydY5?usp=sharing](https://colab.research.google.com/drive/1flsg5PtC8Wlpyq5ynQfO69cGEX_LydY5?usp=sharing).

Nesse link também existem algumas aplicações desenvolvidas em Python que podem auxiliar ainda mais no aprendizado ou para quem já possui algum conhecimento com a linguagem. Esses “tópicos extras” não são abordados durante a disciplina de fundamentos.

## Referências

---

BROCK, D. L. The electronic product code (epc). In: *The Business Professor*. Auto-ID Center White Paper MIT-AUTOIDWH-002, 20018.

FACURE, Matheus. *Aprendendo Representações de Palavras*. 2017. Disponível em: <<https://matheusfacure.github.io/2017/03/20/word2vec/>>. Acesso em: 09 jul. 2020.

LECUN, Yann; CORTES, Corinna; BURGESS, Christopher J.C. *THE MNIST DATABASE of handwritten digits*. Disponível em: <<http://yann.lecun.com/exdb/mnist/>>. Acesso em: 09 jul. 2020.

MONARD, Maria Carolina; BARANAUSKAS, José Augusto. *Conceitos sobre aprendizado de máquina*. Sistemas inteligentes-Fundamentos e aplicações, v. 1, n. 1, p. 32, 2003.

NEDELTCHEV, P. The internet of everything is the new economy. 2015. Disponível em: < [https://www.cisco.com/c/en/us/solutions/collateral/enterprise/cisco-on-cisco/Cisco\\_IT\\_Trends\\_IoE\\_Is\\_the\\_New\\_Economy.html](https://www.cisco.com/c/en/us/solutions/collateral/enterprise/cisco-on-cisco/Cisco_IT_Trends_IoE_Is_the_New_Economy.html)>. Acesso em: 09 jul. 2020.

TIOBE. *TIOBE Index for July 2020*. Disponível em: <<https://www.tiobe.com/tiobe-index/>>. Acesso em: 09 jul. 2020.