



DEFINIÇÃO

Apresentação básica da utilização do Python em outros paradigmas, como linguagem funcional, computação concorrente, desenvolvimento *web* e ciência de dados.

PROpósito

Apresentar a programação por meio do paradigma de linguagem funcional, bem como a utilização do *framework* Flask para aplicações *web* com Python e a sua utilização para a realização de tarefas relacionadas à mineração de dados e à extração de conhecimento.

PREPARAÇÃO

Para este tema, é necessário conhecimento prévio em Python.

OBJETIVOS

MÓDULO 1

Identificar a linguagem funcional e sua utilização em Python

MÓDULO 2

Definir os conceitos de computação concorrente e sua utilização em Python

MÓDULO 3

Identificar o Python como ferramenta para desenvolvimento *web*

MÓDULO 4

Identificar o Python como ferramenta para ciência de dados

INTRODUÇÃO

Neste tema, veremos a utilização da linguagem Python em outros paradigmas, como linguagem funcional, computação concorrente, desenvolvimento *web* e ciência de dados.

Apesar do Python não ser uma linguagem puramente funcional, ela fornece ferramentas para que possamos programar utilizando esse paradigma. Neste tema, veremos como isso é possível e quais as vantagens desse estilo de programação.

Vamos aprender também conceitos relacionados à computação concorrente e à computação paralela, suas diferenças e como podem ser implementadas em Python.

Embora o Python não seja a primeira escolha como linguagem de programação para servidores *web*, é possível encontrar diversos *frameworks* que facilitam, e muito, a sua utilização para tal finalidade. Alguns são minimalistas, como o — Flask — e permitem criar servidores *web* escritos em Python em minutos.

Em contrapartida, essa linguagem de programação é a escolha número 1 para a ciência de dados. Neste tema, iremos conhecer um pouco sobre o processo de extração de conhecimento e como implementá-lo em Python utilizando a biblioteca Scikit-Learn.

Baixe os  **CÓDIGOS FONTE PYTHON**, deste tema, para lhe auxiliar na realização das atividades.

MÓDULO 1

◎ Identificar a linguagem funcional e sua utilização em Python

INTRODUÇÃO

A programação funcional teve seu início no final dos anos 1950, com a linguagem **LISP**.

À diferença do que muitos pensam, esse tipo de programação não é apenas a utilização de funções em seu código-fonte, mas um paradigma e um estilo de programação.

Na programação funcional, toda ação realizada pelo programa deve ser implementada como uma função ou uma composição de funções, mas estas devem seguir algumas regras:

LISP

LISP é uma família de linguagens de programação desenvolvida, em 1958, por John McCarthy. Ela foi pensada a princípio para o processamento de dados simbólicos. Ela é uma linguagem formal matemática.

1

Devem ser puras, ou seja, em qualquer ponto do programa, sempre produzem o mesmo resultado quando passados os mesmos parâmetros;

Os dados devem ser imutáveis, e uma vez definida uma variável, seu valor não pode ser alterado;

2

3

Os *loops* não devem ser utilizados, mas sim a composição de funções ou recursividade.

A utilização dessas regras visa garantir, principalmente, que não haja um **efeito indesejável** e imprevisto quando executamos um programa ou parte dele.

**MUITAS LINGUAGENS DE PROGRAMAÇÃO
(COMO PYTHON, JAVA E C++) DÃO SUPORTE
PARA A PROGRAMAÇÃO FUNCIONAL, PORÉM
SÃO DE PROPÓSITO GERAL, DANDO BASE
PARA OUTROS PARADIGMAS, COMO**

PROGRAMAÇÃO IMPERATIVA E ORIENTADA A OBJETOS.

Outras linguagens, como Haskell, Clojure e Elixir, são predominantemente de programação funcional.

A seguir, ensinaremos como utilizar o Python para programação funcional.

FUNÇÕES PURAS

São aquelas que dependem apenas dos parâmetros de entrada para gerar uma saída. Elas sempre retornam um valor, um objeto ou outra função. Em qualquer ponto do programa, ao chamar uma função pura, com os mesmos parâmetros, devemos obter sempre o mesmo resultado.

Veja os dois *scripts*, funcao1.py e funcao2.py, no **Código 1** seguir:

Código 1 - *Scripts* funcao1.py

```
valor = 20
```

```
def multiplica(fator):
    global valor
    valor = valor * fator
    print("Resultado", valor)
```

```
multiplica(3)
```

```
multiplica(3)
```

Código 1 - *Scripts* funcao2.py

```
valor = 20
```

```
def multiplica(valor, fator):
    valor = valor * fator
    return valor
```

```
print("Resultado", multiplica(valor, 3))
print("Resultado", multiplica(valor, 3))
```

SCRIPT 1

funcao1.py, definimos uma função chamada *multiplica*, que multiplica a variável global *valor* por um **fator** passado como parâmetro. O *valor* do resultado é atribuído à variável *valor* novamente (linha 5), que é impresso em seguida (linha 6).

Ao chamarmos a função *multiplica(3)* pela primeira vez (linha 8), obtemos a saída “*Resultado 60*”. Como modificamos a própria variável global *valor* no corpo da função, ao chamarmos novamente a função *multiplica(3)* (linha 9), obtemos um resultado diferente para a saída: “*Resultado 180*”.

Além de não depender apenas dos parâmetros, essa função não retorna valor algum. A função *multiplica* deste *script* **não** é pura.

SCRIPT 2

funcao2.py, utilizamos a variável **valor** como mais um parâmetro para a função *multiplica*. As duas vezes que executamos essa função (linha 7 e 8), elas retornam o mesmo valor, 60.

A função *multiplica* deste *script* é um exemplo de função pura, pois depende apenas de seus parâmetros para gerar o resultado, e não acessa ou modifica nenhuma variável externa à função e retorna um valor.

DADOS IMUTÁVEIS

São aqueles que não podem ser alterados após sua criação. Apesar do Python disponibilizar algumas estruturas de dados imutáveis, como as tuplas, a maioria é mutável. Na programação funcional, devemos tratar todos os dados como imutáveis!

As funções puras devem utilizar apenas os parâmetros de entrada para gerar as saídas. Além dessa característica, as funções puras não podem alterar nenhuma variável fora de seu escopo.

Observe os *scripts* do **Código 2** a seguir, em que passamos a lista *valores* como argumento para a função *altera_lista*. Lembre-se que, no Python, ao passar uma lista como argumento, apenas passamos sua referência, ou seja, qualquer mudança feita no parâmetro dentro da função, também altera a lista original.

Codigo 2 - *Scripts* funcao3.py

```
valores = [10, 20, 30]
```

```
def altera_lista(lista):
    lista[2] = lista[2] + 10
    return lista

print("Nova lista", altera_lista(valores))
print("Nova lista", altera_lista(valores))
```

Codigo 2 - *Scripts* funcao4.py

```
valores = [10, 20, 30]
```

```
def altera_lista(lista):
    nova_lista = list(lista)
    nova_lista[2] = nova_lista[2] + 10
    return nova_lista

print("Nova lista", altera_lista(valores))
print("Nova lista", altera_lista(valores))
```

Na programação funcional, devemos evitar alterar qualquer dado que já tenha sido criado. No exemplo anterior, no *script* funcao3.py, ao alterar o terceiro elemento do parâmetro *lista* (linha 4), alteramos também a variável global *valores*.

Com isso, ao chamar a mesma função duas vezes (linha 7 e 8), com, teoricamente, o mesmo parâmetro, obtemos um **efeito indesejável**, resultando em saídas diferentes, como no **Código 3**.

Codigo 3 - Saída do *script* funcao3.py

```
C:\Users\ftoli\PycharmProjects\estac
Nova lista [10, 20, 40]
Nova lista [10, 20, 50]
```

Process finished with exit code 0

No exemplo do *script* funcao4.py, muito similar ao anterior, ao invés de alterarmos o valor do próprio parâmetro, criamos uma cópia dele (linha 4), sendo assim, não alteramos a variável

valores e obtemos o mesmo resultado para as duas chamadas da função (linha 8 e 9).

A saída desse script está no **Código 4**.

Código 4 - Saída do script funcao4.py

```
C:\Users\ftoli\PycharmProjects\estac
```

```
Nova lista [10, 20, 40]
```

```
Nova lista [10, 20, 40]
```

```
Process finished with exit code 0
```

EFEITO COLATERAL E ESTADO DA APLICAÇÃO

As funções puras e dados imutáveis buscam evitar os efeitos indesejáveis, como ocorreu no script funcao3.py. Na terminologia de programação funcional, chamamos isso de **efeito colateral** (*side effect*). Além de evitar o efeito colateral, a programação funcional evita a **dependência do estado** de um programa.

A dependência apenas dos parâmetros para gerar saídas garante que o resultado será sempre o mesmo, independentemente do estado da aplicação, por exemplo, valores de outras variáveis. Ou seja: não teremos diferentes comportamentos para uma função baseado no estado atual da aplicação.

EFEITO COLATERAL

O **efeito colateral** é quando a função faz alguma operação que não seja computar a saída a partir de uma entrada. Como por exemplo: alterar uma variável global, escrever no console, alterar um arquivo, inserir um registro no banco de dados, ou enviar um foguete à lua.

DICA

Ao garantir que uma função utilizará apenas os dados de entrada para gerar um resultado e que nenhuma variável fora do escopo da função será alterada, temos a certeza de que não teremos um problema escondido, ou efeito colateral em nenhuma outra parte do código.

O objetivo principal da programação funcional não é utilizar funções puras e dados imutáveis, mas sim evitar o efeito colateral.

FUNÇÕES DE ORDEM SUPERIOR

Na programação funcional, é muito comum utilizar funções que aceitem outras funções, como parâmetros ou que retornem outra função.

Essas funções são chamadas de funções de ordem superior (*higher order function*).

No exemplo do **Código 5** a seguir, vamos criar uma função de ordem superior chamada *multiplicar_por*. Ela será utilizada para criar e retornar novas funções.

Essa função, ao ser chamada com um determinado **multiplicador** como argumento, retorna uma nova função multiplicadora por aquele **multiplicador** e que tem como parâmetro o número a ser multiplicado (**multiplicando**).

Código 5 - Scripts funcao5.py

```
def multiplicar_por(multiplicador):
    def multi(multiplicando):
        return multiplicando * multiplicador
    return multi

multiplicar_por_10 = multiplicar_por(10)
print(multiplicar_por_10(1))
print(multiplicar_por_10(2))
```

```
multiplicar_por_5 = multiplicar_por(5)
print(multiplicar_por_5(1))
print(multiplicar_por_5(2))
```

Dentro da função “pai” *multiplicar_por*, definimos a função *multi* (linha 2), que espera um parâmetro chamado **multiplicando**, que será multiplicado pelo **multiplicador** passado como parâmetro para a função “pai”.

Ao chamar a função *multiplicar_por* com o argumento 10 (linha 6), definimos a função interna *multi* como:

DEF MULTI(MULTIPLICANDO):

RETURN MULTIPLICANDO * 10

Essa função é retornada e armazenada na variável *multiplicar_por_10* (linha 6), que nada mais é que uma referência para a função *multi* recém-criada.

Dessa forma, podemos chamar a função *multiplicar_por_10*, passando um número como argumento, o multiplicando, para ser multiplicado por 10 (linhas 7 e 8). Produzindo os resultados 10 e 20.

Da mesma forma, criamos a função *multiplicar_por_5*, passando o número 5 como argumento para a função *multiplicar_por* (linha 10), que recebe uma referência para a função:

DEF MULTI(MULTIPLICANDO):

RETURN MULTIPLICANDO * 5

Com isso, podemos utilizar a função *multiplicar_por_5* para multiplicar um número por 5 (linhas 11 e 12).

Observe a saída do programa no console abaixo.

Codigo 5 - Saída do script funcao5

```
C:\Users\fotoli\PycharmProjects\estac
```

```
10  
20  
5  
10
```

```
Process finished with exit code 0
```

FUNÇÕES LAMBDA

Assim como em outras linguagens, o Python permite a criação de funções anônimas. Estão são definidas sem identificador (ou nome) e, normalmente, são utilizadas como argumentos para outras funções (de ordem superior).

Em Python, as funções anônimas são chamadas de **funções lambda**. Para criá-las, utilizamos a seguinte sintaxe:

LAMBDA ARGUMENTOS: EXPRESSÃO

Iniciamos com a palavra reservada **lambda**, seguida de uma sequência de argumentos separados por vírgula, dois pontos e uma expressão de apenas uma linha. As funções **lambda** SEMPRE retornam o valor da expressão automaticamente. Não é necessário utilizar a palavra **return**.

★ EXEMPLO

Considere a função para multiplicar dois números a seguir:

```
def multiplicar(a, b):  
    return a*b
```

A função *lambda* equivalente é:

```
lambda a, b: a*b
```

Temos os parâmetros **a** e **b** e a expressão **a*b**, que é retornado automaticamente. As funções *lambda* podem ser armazenadas em variáveis para depois serem chamadas como uma função qualquer.

Retornando ao exemplo da função *multiplicar_por*, podemos trocar a função *multi* por uma função *lambda*. Observe no **Código 6**, a seguir, em que abaixo (*anonima.py*), temos a utilização da função *lambda* e após a utilização da função original (*funcao5.py*):

Código 6 - Script *anonima.py*

```
def multiplicar_por(multiplicador):  
    return lambda multiplicando: multiplicando * multiplicador
```

```
multiplicar_por_10 = multiplicar_por(10)  
print(multiplicar_por_10(1))  
print(multiplicar_por_10(2))
```

```
multiplicar_por_5 = multiplicar_por(5)  
print(multiplicar_por_5(1))  
print(multiplicar_por_5(2))
```

Código 6 - Script *funcao5.py*

```
def multiplicar_por(multiplicador):  
    def multi(multiplicando):  
        return multiplicando * multiplicador  
    return multi
```

```
multiplicar_por_10 = multiplicar_por(10)  
print(multiplicar_por_10(1))  
print(multiplicar_por_10(2))
```

```
multiplicar_por_5 = multiplicar_por(5)  
print(multiplicar_por_5(1))  
print(multiplicar_por_5(2))
```

Ao executar o script *anonima.py*, obtemos o mesmo resultado mostrado anteriormente: 10, 20, 5 e 10.

NÃO UTILIZAR LOOPS

Outra regra, ou boa prática, da programação funcional é não utilizar laços (*for* e *while*), mas sim composição de funções ou recursividade.

A função *lambda* exerce um papel fundamental nisso, como veremos a seguir.

Para facilitar a composição de funções e evitar *loops*, o Python disponibiliza diversas funções e operadores.

As funções internas mais comuns são *map* e *filter*.

MAP

A função *map* é utilizada para aplicar uma determinada função em cada elemento de um *iterável* (lista, tupla, dicionários etc.), retornando um novo **iterável** com os valores modificados.

A FUNÇÃO MAP É PURA E DE ORDEM SUPERIOR, POIS DEPENDE APENAS DE SEUS PARÂMETROS E RECEBE UMA FUNÇÃO COMO PARÂMETRO. A SUA SINTAXE É A SEGUINTE:

MAP(FUNÇÃO, ITERÁVEL1, ITERÁVEL2...)

O primeiro parâmetro da *map* é o nome da função (sem parênteses) que será executada para cada item do iterável. Os demais parâmetros são os iteráveis separados por vírgula. A função *map* SEMPRE retorna um novo iterável.

No exemplo do **Código 7**, a seguir, vamos criar três *scripts*. Todos executam a mesma operação. Recebem uma lista e triplicam cada item, gerando uma **nova** lista com os valores

triplicados.

Codigo 7 - Script *funcao_iterable.py*

```
lista = [1, 2, 3, 4, 5]
```

```
def triplica_itens(iterable):
```

```
    lista_aux = []
```

```
    for item in iterable:
```

```
        lista_aux.append(item*3)
```

```
    return lista_aux
```

```
nova_lista = triplica_itens(lista)
```

```
print(nova_lista)
```

Codigo 7 - Script *funcao_map.py*

```
lista = [1, 2, 3, 4, 5]
```

```
def triplica(item):
```

```
    return item * 3
```

```
nova_lista = map(triplica, lista)
```

```
print(list(nova_lista))
```

Codigo 7 - Script *funcao_map_lambda.py*

```
lista = [1, 2, 3, 4, 5]
```

```
nova_lista = map(lambda item: item * 3, lista)
```

```
print(list(nova_lista))
```

PRIMEIRO SCRIPT

funcao_iterable.py, definimos uma função chamada *triplica_item*, que recebe um **iterável** como parâmetro (linha 3), cria uma lista auxiliar para garantir imutabilidade (linha 4), percorre os itens do iterável passado como parâmetro (linha 5), adiciona os itens triplicados à lista auxiliar (linha 6) e retorna a lista auxiliar (linha 7).

Essa função é chamada com o argumento *lista* (linha 9) e o resultado é impresso (linha 10).

SEGUNDO SCRIPT

funcao_map.py, definimos a função *triplica*, que triplica e retorna o item passado como parâmetro (linha 3 e 4). É utilizada, assim como a variável *lista*, como argumentos para a função *map* (linha 6).

A *map* vai aplicar internamente a função passada como parâmetro em cada item da *lista*, retornando um novo **iterável** (que pode ser convertido em listas, tuplas etc.). O resultado da função *map* é armazenado na variável *nova_lista*, para então ser impresso (linha 7).

A função *map* garante a imutabilidade dos **iteráveis** passados como argumento. Como a função *map* retorna um **iterável**, utilizamos o construtor *list(iterável)* para imprimir o resultado (linha 7).

TERCEIRO SCRIPT

funcao_map_lambda.py, substituímos a função *triplica* pela função *lambda* (*lambda item: item*3*), que foi utilizada como argumento para a função *map* (linha 3). Esta vai aplicar a função *lambda* em cada item da *lista*, retornando um novo **iterável** que foi impresso posteriormente (linha 4).

Observe como o código foi reduzido e mesmo assim preservamos a utilização de funções puras (*lambda*), alta ordem (*map*) e que preservaram a imutabilidade dos dados. Tudo isso para garantir que não haja efeitos colaterais e dependência de estados.

Todos os *scripts* utilizam funções puras e geraram o mesmo resultado: [3, 6, 9, 12, 15].

FILTER

É utilizada para filtrar elementos de um **iterável** (lista, tupla, dicionários etc.). O filtro é realizado utilizando uma função, que deve ser capaz de retornar **true ou false** para cada elemento do *iterável*.

TRUE OU FALSE

Verdadeiro ou falso.

ATENÇÃO

Todo elemento que for avaliado como *true* será incluído em um novo **iterável** retornado pela função *filter*, que é pura e de alta ordem, pois depende apenas dos parâmetros e recebe uma função como parâmetro. A sua sintaxe é a seguinte:

filter(função, iterável)

O primeiro parâmetro da função *filter* é o nome da função (sem parênteses), que será executada para cada item do **iterável**. O segundo parâmetro é o **iterável**. A função *filter* SEMPRE retorna um novo **iterável**, mesmo que vazio.

No exemplo do **Código 8** a seguir, vamos criar três *scripts*. Todos fazem a mesma filtragem. Recebem uma lista e retornam os elementos ímpares, gerando uma nova lista, de forma a garantir a imutabilidade.

Código 8 - *Script* funcao_filtro_iterable.py

```
lista = [1, 2, 3, 4, 5]
```

```
def impares(iterable):
    lista_aux = []
    for item in iterable:
        if item % 2 != 0:
            lista_aux.append(item)
    return lista_aux
```

```
nova_lista = impares(lista)
print(nova_lista)
```

Código 8 - *Script* funcao_filter.py

```
lista = [1, 2, 3, 4, 5]
```

```
def impar(item):
    return item % 2 != 0
```

```
nova_lista = filter(impar, lista)
print(list(nova_lista))
```

Código 8 - *Script* funcao_filter_lambda.py

```
lista = [1, 2, 3, 4, 5]
```

```
nova_lista = filter(lambda item: item % 2 != 0, lista)  
print(list(nova_lista))
```

SCRIPT FUNÇÃO_FILTERABLE.PY

Definimos uma função chamada **ímpares**, que recebe um **iterável** como parâmetro (linha 3), cria uma lista auxiliar para garantir imutabilidade (linha 4), percorre os itens do **iterável** passados como parâmetros (linha 5), adiciona os itens ímpares à lista auxiliar (linhas 6 e 7) e retorna a *lista* auxiliar (linha 8).

Essa função é chamada com o argumento *lista* (linha 10) e o resultado é impresso (linha 11).

SCRIPT FUNCAO_FILTER.PY

Definimos a função *ímpar*, que retorna *true* se o item passado como parâmetro é ímpar ou *false* caso contrário (linhas 3 e 4). Utilizamos essa função, assim como a variável *lista*, como argumentos para a função *filter* (linha 6).

A *filter* vai aplicar, internamente, a função passada como parâmetro em cada item da *lista*, retornando um novo **iterável** (que pode ser convertido em listas, tuplas etc.). O resultado da função *filter* é armazenado na variável *nova_lista*, para então ser impresso (linha 7).

A função *filter* garante a imutabilidade dos **iteráveis** passados como argumento. Como a função *filter* retorna um **iterável**, utilizamos o construtor *list(iterável)* para imprimir o resultado (linha 7).

SCRIPT, FUNÇÃO_FILTER_LAMBDA.PY

Substituímos a função *ímpar* pela função *lambda* (*lambda item: item%2 != 0*) que foi utilizada como argumento para a função *filter* (linha 3). Esta vai aplicar a função *lambda* em cada item da *lista*, retornando um novo **iterável** que foi impresso posteriormente (linha 4).

Todos os *scripts* geraram o mesmo resultado: [1, 3, 5].

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. OBSERVE AS AFIRMATIVAS RELACIONADAS À PROGRAMAÇÃO FUNCIONAL E RESPONDA.

- I - AS FUNÇÕES PURAS SEMPRE PRODUZEM O MESMO RESULTADO QUANDO PASSADOS OS MESMOS PARÂMETROS.
- II - DADOS IMUTÁVEIS SÃO AQUELES NOS QUAIS SEUS VALORES PODEM SER ALTERADOS APÓS A SUA DEFINIÇÃO.
- III - NÃO SE DEVE UTILIZAR *LOOPS*, MAS COMPOSIÇÃO DE FUNÇÕES.
- IV - A PROGRAMAÇÃO FUNCIONAL É UM PARADIGMA E UM ESTILO DE PROGRAMAÇÃO.

DAS AFIRMATIVAS ANTERIORES, QUAIS SÃO VERDADEIRAS?

- A) II e III.
- B) I e III.
- C) II.
- D) I, III e IV.

2. QUAL É O RESULTADO IMPRESSO PELO PROGRAMA A SEGUIR?

SCRIPT LAMBDA1.PY

MINHA_FUNCAO = LAMBDA X: X ** 2

RESULTADO = MINHA_FUNCAO(4)

PRINT(RESULTADO)

A) 4.

B) 8.

C) 16.

D) 32.

GABARITO

1. Observe as afirmativas relacionadas à programação funcional e responda.

I - As funções puras sempre produzem o mesmo resultado quando passados os mesmos parâmetros.

II - Dados imutáveis são aqueles nos quais seus valores podem ser alterados após a sua definição.

III - Não se deve utilizar *loops*, mas composição de funções.

IV - A programação funcional é um paradigma e um estilo de programação.

Das afirmativas anteriores, quais são verdadeiras?

A alternativa "D" está correta.

A única incorreta é a II. Os dados imutáveis NÃO podem ser alterados.

2. Qual é o resultado impresso pelo programa a seguir?

Script lambda1.py

minha_funcao = lambda x: x ** 2

resultado = minha_funcao(4)

print(resultado)

A alternativa "C" está correta.

A função criada retorna o valor do argumento ao quadrado.

MÓDULO 2

-
- Definir os conceitos de computação concorrente e sua utilização em Python

INTRODUÇÃO

No início do ano 2000, poucas pessoas tinham acesso a computadores com mais de um processador. Além disso, os processadores dos *desktops* e *notebooks* daquela época continham apenas um núcleo, ou seja, só podiam executar uma operação por vez.



Fonte: Nonchanon/Shutterstock

Atualmente, é raro ter um computador, ou até mesmo um celular, com um processador de apenas um núcleo. Com isso, é muito importante aprendermos alguns conceitos sobre programação concorrente e paralela e como implementá-los, de forma a extrair o máximo de performance dos nossos programas.

CONCORRENTE

Alerta! Clique no item acima. ×

Na computação, temos dois conceitos relacionados à execução simultânea, **concorrência** e **paralelismo**, que serão apresentados no decorrer deste módulo.

A seguir, definiremos alguns conceitos relacionados à computação concorrente e paralela, e, posteriormente, mostraremos como o Python implementa tais funcionalidades.

CONCEITOS

PROGRAMAS E PROCESSOS

Os conceitos iniciais que precisamos definir são **programas** e **processos**. Mas o que seria um programa e um processo na visão do sistema operacional?

PROGRAMA É ALGO ESTÁTICO E PERMANENTE. CONTÉM UM CONJUNTO DE INSTRUÇÕES E É PERCEBIDO PELO SISTEMA OPERACIONAL COMO PASSIVO.

Em contrapartida, o **processo** é dinâmico e tem uma vida curta. Ao longo da sua execução, tem seu estado alterado. Ele é composto por código, dados e contexto. Em suma, o processo é um programa em execução.

O mesmo programa executado mais de uma vez gera processos diferentes, que contêm dados e momento de execução (contexto) também diferentes.

CONCORRÊNCIA E PARALELISMO

Em computação, o termo **concorrente** se refere a uma técnica para tornar os programas mais usáveis. Ela pode ser alcançada mesmo em processadores de apenas um núcleo, pois permite compartilhar o processador de forma a rodar vários processos. Os sistemas operacionais multitarefas suportam concorrência.

★ EXEMPLO

Por exemplo, é possível compactar um arquivo e continuar usando o processador de texto, mesmo em um processador de um núcleo, pois o processador divide o tempo de uso entre os processos dessas duas aplicações. Ou seja: essas duas aplicações executam de forma concorrente.

Apesar da concorrência não necessariamente precisar de mais de um núcleo, ela pode se beneficiar se houver mais deles, pois os processos não precisarão compartilhar tempo de processador.

Já o **paralelismo** é uma técnica para permitir que programas rodem mais rápido, executando várias operações ao mesmo tempo. Ela requer várias máquinas ou um processador com mais de um núcleo. Também é possível realizar o paralelismo nas placas de vídeo.

THREADS E PROCESSOS

⌚ RELEMBRANDO

O **processo** é uma instância de um programa em execução. Ele é composto pelo código que está sendo executado, os dados utilizados pelo programa (exemplo: valores de variável) e o estado ou contexto do processo.

Em um programa com múltiplos processos, cada um tem seu próprio espaço de memória e cada um pode executar em um núcleo diferente, melhorando consideravelmente a performance do programa.

② VOCÊ SABIA?

A **thread** pertence a um determinado processo. Múltiplas *threads* podem ser executadas dentro de um mesmo processo. As de um mesmo processo compartilham a área de memória e podem acessar os mesmos dados de forma transparente.

Em Python, podemos criar tanto processos quanto *threads* em um programa.

Antes de falarmos sobre cada abordagem em Python, vamos falar sobre o *global interpreter lock* (GIL). No CPython — que é a implementação principal da linguagem de programação Python, escrita em linguagem C — o GIL existe para proteger o acesso aos objetos da linguagem.



Fonte: Nonchanon/Shutterstock

Isso acontece objetivando a prevenção para que múltiplas *threads* não possam executar os *bytecodes* do Python de uma vez. Essa “trava” é necessária, pois visa garantir a integridade do interpretador, uma vez que o gerenciamento de memória no CPython não é *thread-safe*.

NA PRÁTICA, PARA UM MESMO PROCESSO, O GIL SÓ PERMITE EXECUTAR UMA *THREAD* DE CADA VEZ, OU SEJA, ELAS NÃO EXECUTAM DE FORMA PARALELA, MAS CONCORRENTES. SEM A “TRAVA” DO GIL, UMA OPERAÇÃO SIMPLES DE ATRIBUIÇÃO DE VARIÁVEL PODERIA GERAR UM DADO INCONSISTENTE CASO DUAS *THREADS* ATRIBUÍSSEM UM VALOR A UMA MESMA VARIÁVEL AO MESMO TEMPO.

Para os processos, por sua vez, o funcionamento é diferente. Para cada um, temos um GIL separado. Ou seja: podem ser executados paralelamente. Cabe ao programador garantir o acesso correto aos dados.

Múltiplos processos podem rodar em paralelo, enquanto múltiplas *threads* (de um mesmo processo) podem rodar concorrentemente.

ATENÇÃO

Normalmente, utilizamos *thread* para interface gráfica, acesso ao banco de dados, acesso à rede etc., pois o programa não pode parar, ou a interface congelar, enquanto esperamos baixar um arquivo, por exemplo.

Porém, quando temos um programa que necessita muito dos recursos do processador e temos como paralelizar nosso programa, devemos optar pela criação de múltiplos processos.

Criar novos processos pode ser lento e consumir muita memória, enquanto a criação de *threads* é mais rápida e consome menos memória.

CRIAÇÃO DE *THREADS* E PROCESSOS

★ EXEMPLO

Como criar *threads* e processos em Python:

Vamos utilizar a classe *thread* e *process*, dos módulos *threading* e *multiprocessing*, respectivamente. Para facilitar a transição entre processos e *threads* simples, o Python fez os construtores e métodos das duas classes muito parecidos.

No *script principal.py* do **Código 9**, vamos criar uma *thread* e um processo que executam a mesma função. Na linha 8, criamos uma *thread* para executar a função *funcao_a* utilizando a classe *thread*. O construtor tem como parâmetros a função que deverá ser executada (*target*) e quais parâmetros serão passados para essa função (*args*). O parâmetro *args* espera um *iterável* (lista, tupla etc.), onde cada elemento será mapeado para um parâmetro da função *target*.

Como a *funcao_a* espera apenas um parâmetro, definimos uma tupla de apenas um elemento ('Minha Thread'). O primeiro elemento da tupla, a *string* Minha Thread, será passada para o parâmetro nome da *funcao_a*.

Na linha 9, enviamos o comando para a *thread* iniciar sua execução, chamando o método *start()* do objeto *t* do tipo *thread*.

Código 9 - *Script principal.py*

```
from threading import Thread
from multiprocessing import Process

def funcao_a(nome):
    print(nome)

if __name__ == '__main__':
    t = Thread(target=funcao_a, args=("Minha Thread",))
    t.start()

p = Process(target=funcao_a, args=("Meu Processo",))
p.start()
```

A criação do processo é praticamente igual, porém utilizando a classe *process*, conforme a linha 11. Para iniciar o processo, chamamos o método *start()* na linha 12.

Verifique a saída desse *script* no console abaixo.

Codigo 9 - Saída do *script* principal(1)

```
C:\Users\ftoli\PycharmProjects\estac
```

Minha Thread

Meu Processo

Process finished with exit code 0

MÚLTIPLAS *THREADS* E PROCESSOS

No exemplo do **Codigo 10** a seguir, vamos criar múltiplas *threads* e processos usando na criação de para criar *threads* e processos para compararmos as saídas de cada um.

VAMOS APROVEITAR PARA MOSTRAR QUE TODAS AS *THREADS* ESTÃO NO MESMO CONTEXTO, COM ACESSO ÀS MESMAS VARIÁVEIS, ENQUANTO O PROCESSO NÃO. VAMOS MOSTRAR, TAMBÉM, COMO FAZER O PROGRAMA AGUARDAR QUE TODAS AS *THREADS* E PROCESSOS TERMINEM ANTES DE SEGUIR A EXECUÇÃO.

► ATENÇÃO

Observe que os códigos dos *scripts* são praticamente idênticos, com exceção das classes construtoras *thread* e *process* na linha 16 dos dois *scripts*.

Neste exemplo, a função que será paralelizada é a *funcao_a* (linha 7). Ela contém um laço que é executado cem mil vezes e para cada iteração adiciona o elemento 1 à lista *minha_lista*,

definida globalmente na linha 5.

Vamos criar 10 *threads* (e processos) para executar 10 instâncias dessa função, na qual, esperamos que o número de elementos na lista ao final da execução do programa seja de 1 milhão (10 *threads* X cem mil iterações).

Para criar essas 10 *threads* ou processos, temos um laço de 10 iterações (linha 15), em que criamos (linha 16) e iniciamos (linha 18) cada *thread* ou processo.

O OBJETIVO DA CRIAÇÃO DE *THREADS* OU PROCESSOS É JUSTAMENTE A COMPUTAÇÃO CONCORRENTE OU PARALELA, O PYTHON NÃO FICA “PARADO” AGUARDANDO DELA APÓS CHAMAR O MÉTODO *START()*, ELE IMEDIATAMENTE COMEÇA A PRÓXIMA ITERAÇÃO DO LAÇO *FOR*.

Codigo 10 - Script threads_var.py

```
from threading import Thread, Lock
from multiprocessing import Process
import time
```

```
minha_lista = []
```

```
def funcao_a(indice):
    for i in range(100000):
        minha_lista.append(1)
    print("Termino thread ", indice)
```

```
if __name__ == '__main__':
    tarefas = []
    for indice in range(10):
        tarefa = Thread(target=funcao_a, args=(indice,))
```

```
tarefas.append(tarefa)
tarefa.start()

print("Antes de aguardar o termino!", len(minha_lista))

for tarefa in tarefas:
    tarefa.join()

print("Após aguardar o termino!", len(minha_lista))
```

Codigo 10 - *Script* processos_var.py

```
from threading import Thread, Lock
from multiprocessing import Process
import time

minha_lista = []

def funcao_a(indice):
    for i in range(100000):
        minha_lista.append(1)
    print("Termino processo ", indice)

if __name__ == '__main__':
    tarefas = []
    for indice in range(10):
        tarefa = Process(target=funcao_a, args=(indice,))
        tarefas.append(tarefa)
        tarefa.start()

    print("Antes de aguardar o termino!", len(minha_lista))

    for tarefa in tarefas:
        tarefa.join()

    print("Após aguardar o termino!", len(minha_lista))
```

ATENÇÃO

É função do programador armazenar uma referência para as suas *threads* ou processos, de maneira que possamos verificar seu estado ou interrompê-las. Para isso, armazenamos cada *thread* ou processo criado em uma lista chamada *tarefas* (linha 17).

Logo após a criação das *threads* ou processos, vamos imprimir o número de itens na variável *minha_lista* (linha 20); aguardar o término da execução das *threads* ou processos pela iteração da lista *tarefas* e utilizando o método *join()* (linhas 22 e 23); e, por fim, imprimimos o número de itens final da variável *minha_lista* (linha 25).

No **Código 11**, a seguir, temos o resultado de cada *script*.

Código 11 - Saída do *script* threads_var.py

```
C:\Users\ftoli\PycharmProjects\estacio_ead\venv\Scripts\python.e
```

```
Termino thread Termino thread 0 1
```

```
Termino thread Termino thread 23Termino thread 4
```

```
Termino thread 5
```

```
Termino thread 6
```

```
Termino thread 7
```

```
Termino thread Antes de guardar o termino! Termino thread 9
```

```
970012
```

```
8
```

```
Apos guardar o termino! 1000000
```

```
Process finished with exit code 0
```

Código 11 - Saída do *script* processos_var.py

```
C:\Users\ftoli\PycharmProjects\estacio_ead\venv\
```

```
Antes guardar o termino! 0
```

```
Termino thread 0
```

```
Termino thread 2
```

```
Termino thread 1
```

```
Termino thread 4
```

```
Termino thread 3
```

```
Termino thread 5
Termino thread 7
Termino thread 6
Termino thread 8
Termino thread 9
Apos guardar o termino! 0
```

Process finished with exit code 0

Apesar da saída do *script thread.py* ter ficado confusa (já iremos retornar nesse problema), observe que o número de itens da variável *minha_lista* muda durante a execução do programa quando usamos *thread* e não muda quando usamos processos.

Isso acontece, pois a área de memória das *threads* é compartilhada, ou seja, elas têm acesso às mesmas variáveis globais. Em contrapartida, as áreas de memória dos processos não são compartilhadas. Cada processo acaba criando uma versão própria da variável *minha_lista*.

TRAVAS (LOCK)

No exemplo anterior, a função *funcao_a* incluía o elemento 1 à lista a cada iteração, utilizando o método *append()*. No exemplo a seguir, a *funcao_a* irá incrementar a variável global *contador*. Observe o código do **Código 12** e verifique o resultado impresso no console seguinte.

Código 12 - Script threads_inc.py

```
from threading import Thread, Lock
from multiprocessing import Process
import time
```

```
contador = 0
```

```
def funcao_a(indice):
    global contador
    for i in range(1000000):
        contador += 1
    print("Termino thread ", indice)
```

```
if __name__ == '__main__':
    tarefas = []
    for indice in range(10):
        tarefa = Thread(target=funcao_a, args=(indice,))
        tarefas.append(tarefa)
        tarefa.start()
```

```
print("Antes de aguardar o termino!", contador)
```

```
for tarefa in tarefas:
    tarefa.join()
```

```
print("Após aguardar o termino!", contador)
```

Codigo 12 - Saída do *script threads_inc*

```
C:\Users\ftoli\PycharmProjects\estacio_ead\venv\S
```

```
Termino thread 1
```

```
Termino thread Termino thread 3 2
```

```
Termino thread 0 Termino thread
```

```
Antes de aguardar o termino! Termino thread 4
```

```
5Termino thread 6
```

```
2235554
```

```
Termino thread Termino thread 9
```

```
Termino thread 7
```

```
8
```

```
Apos aguardar o termino! 3316688
```

```
Process finished with exit code 0
```

O valor impresso ao final do programa deveria ser 10.000.000 (10 *threads* X 1.000.000 de iterações), porém foi impresso 3.316.688. Você deve estar se perguntando por que aconteceu isso, se o GIL garante que apenas uma *thread* esteja em execução por vez.

E POR QUE NÃO ACONTECEU ISSO NO EXEMPLO ANTERIOR?

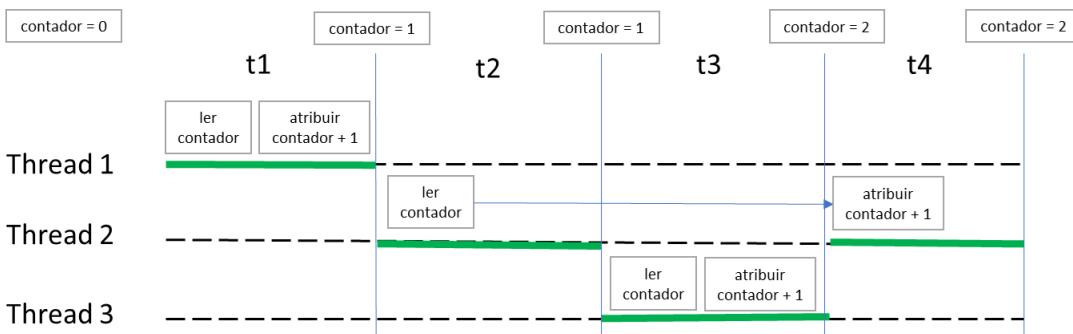
RESPOSTA

E POR QUE NÃO ACONTECEU ISSO NO EXEMPLO ANTERIOR?

Vamos começar pelo exemplo anterior. O método utilizado para inserir um elemento na lista (*append*), na visão do GIL, é atômico, ou seja, ele é executado de forma segura e não pode ser interrompido no meio de sua execução.

O incremento de variável ($+1$), que está sendo usado no exemplo atual (linha 10), não é atômico. Ele é, na verdade, composto por duas operações, a leitura e a atribuição, e não temos como garantir que as duas operações serão realizadas atomicamente.

Veja a **figura 1**, em que temos um contador inicialmente com o valor 0 e três *threads* incrementando esse contador. Para incrementar, a *thread* realiza duas operações: lê o valor do contador e depois atribui o valor lido incrementado de um ao contador.



Fonte: Autor

Figura 1.

Cada *thread* é executada em um determinado tempo t. Em t1, a *thread1* lê e incrementa o contador, que passa a valer 1. Em t2, a *thread2* lê o contador (valor 1). Em t3, a *thread3* lê e

incrementa o contador, que passa a valer 2. Em t4, a *thread2* incrementa o contador, porém a partir do valor que ela leu, que era 1.

No final, o contador passa a valer 2, erroneamente. Esse resultado inesperado devido à sincronia na concorrência de *threads* (ou processos) se chama **condição de corrida**.

Para evitar a condição de corrida, utilizamos a primitiva *lock* (trava). Um objeto desse tipo tem somente dois estados: travado e destravado. Quando criado, ele fica no estado destravado. Esse objeto tem dois métodos: *acquire* e *release*.

Quando no estado destravado, o *acquire* muda o estado dele para travado e retorna imediatamente. Quando no estado travado, o *acquire* bloqueia a execução até que outra *thread* faça uma chamada ao método *release* e destrave o objeto. Veja como adaptamos o código anterior para utilizar o *lock* no **Código 13**.

Código 13 - Script threads_inc_lock.py

```
from threading import Thread, Lock
from multiprocessing import Process
import time

contador = 0
lock = Lock()
print_lock = Lock()

def funcao_a(indice):
    global contador
    for i in range(1000000):
        lock.acquire()
        contador += 1
        lock.release()
        print_lock.acquire()
        print("Termino thread ", indice)
        print_lock.release()

if __name__ == '__main__':
    tarefas = []
    for indice in range(10):
        tarefa = Thread(target=funcao_a, args=(indice,))
        tarefas.append(tarefa)
        tarefa.start()
    for tarefa in tarefas:
        tarefa.join()
```

```
tarefas.append(tarefa)
tarefa.start()

print("Antes de aguardar o termino!", contador)

for tarefa in tarefas:
    tarefa.join()

print("Após aguardar o termino!", contador)
```

Primeiramente, definimos a variável global *lock* utilizando o construtor *lock()*, também do módulo *threading* (linha 6). Depois, vamos utilizar essa trava para “envolver” a operação de incremento. Imediatamente antes de incrementar o contador, chamamos o método *acquire()* da variável *lock* (linha 12), de forma a garantir exclusividade na operação de incremento (linha 13).



Logo após o incremento, liberamos a trava utilizando o método *release* (linha 14). Com apenas essa modificação, garantimos que o resultado estará correto. Podemos verificar o resultado no console abaixo.

Código 13 - Saída do script *threads_inc_lock*

```
C:\Users\ftoli\PycharmProjects\estacio
```

```
Antes de aguardar o termino! 73844
```

```
Termino thread 1
```

```
Termino thread 7
```

```
Termino thread 8
```

```
Termino thread 9
```

```
Termino thread 2
```

```
Termino thread 3
```

```
Termino thread 0
```

```
Termino thread 4
```

```
Termino thread 6
```

```
Termino thread 5
```

```
Após aguardar o termino! 10000000
```

```
Process finished with exit code 0
```

Aproveitamos este exemplo para corrigir o problema de impressão no console de forma desordenada. Esse problema ocorria, pois o *print* também não é uma operação atômica. Para resolver, envolvemos o *print* da linha 16 com outra trava, *print_lock*, criada na linha 7.

COMPARTILHANDO VARIÁVEIS ENTRE PROCESSOS

Para criar uma área de memória compartilhada e permitir que diferentes processos acessem a mesma variável, podemos utilizar a classe *Value* do módulo *multiprocessing*.

No exemplo do **Código 14** a seguir, vamos criar uma variável do tipo inteiro e compartilhá-la entre os processos. Essa variável fará o papel de um contador e a função paralelizada vai incrementá-la.

Código 14 - Script *processos.py*

```
from threading import Thread
from multiprocessing import Process, Value

def funcao_a(indice, cont):
    for i in range(100000):
        with cont.get_lock():
            cont.value += 1
        print("Termino processo ", indice)

if __name__ == '__main__':
    contador = Value('i', 0)
    tarefas = []
    for indice in range(10):
        tarefa = Process(target=funcao_a, args=(indice, contador))
        tarefas.append(tarefa)
        tarefa.start()

    print("Antes de aguardar o termino!", contador.value)

    for tarefa in tarefas:
```

```
tarefa.join()
```

```
print("Após aguardar o termino!", contador.value)
```

Codigo 14 - Saída do *script* processos

```
C:\Users\ftoli\PycharmProjects\estacio_e
```

```
Antes de aguardar o termino! 0
```

```
Termino processo 2
```

```
Termino processo 1
```

```
Termino processo 0
```

```
Termino processo 3
```

```
Termino processo 4
```

```
Termino processo 5
```

```
Termino processo Termino processo 7
```

```
6
```

```
Termino processo 8
```

```
Termino processo 9
```

```
Após aguardar o termino! 1000000
```

Process finished with exit code 0

Para permitir que a variável seja compartilhada, declaramos uma variável chamada *contador* utilizando o construtor da classe *value*, onde passamos como primeiro argumento um caractere com o tipo da variável compartilhada ('i' → inteiro) e seu valor inicial (0) (linha 11).

**COMO NÃO TEMOS ACESSO A VARIÁVEIS
GLOBAIS ENTRE OS PROCESSOS,
PRECISAMOS PASSAR ESTA PARA O
CONSTRUTOR *PROCESS* POR MEIO DO
PARÂMETRO *ARGS*. COMO A PASSAGEM DE
PARÂMETROS É POSICIONAL, O PARÂMETRO
INDICE DA FUNCAO_A RECEBE O VALOR DA**

VARIÁVEL ÍNDICE E O PARÂMETRO `CONT` RECEBE UMA REFERÊNCIA PARA A VARIÁVEL CONTADOR (LINHA 14).

Isso já é o suficiente para termos acesso a variável `contador` por meio do parâmetro `cont` da função `funcao_a`. Porém, não resolve a condição de corrida.

Para evitar esse problema, vamos utilizar uma trava (*lock*) para realizar a operação de incremento, assim como fizemos no exemplo anterior.

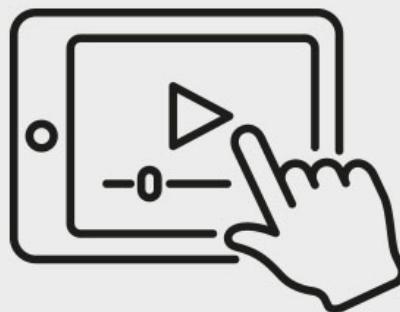
② VOCÊ SABIA?

O Python já disponibiliza essa trava nos objetos do tipo `value`, não sendo necessário criar uma variável específica para a trava (*lock*). Observe como foi realizada a trava utilizando o método `get_lock()` (linha 6).

Observe pelo console que agora nossa variável está com o valor certo: um milhão!

Para corrigir o problema da impressão desordenada, basta criar uma variável do tipo `lock` e passá-la como parâmetro, assim como fizemos no exemplo anterior e com a variável `contador`.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. OBSERVE AS AFIRMATIVAS E RESPONDA:

- I – É POSSÍVEL ALCANÇAR A CONCORRÊNCIA EM PROCESSADORES DE APENAS UM NÚCLEO.**
- II – O PARALELISMO É UMA TÉCNICA PARA PERMITIR EXECUTAR VÁRIAS OPERAÇÕES AO MESMO TEMPO.**
- III – PROGRAMAS E PROCESSOS TÊM A MESMA DEFINIÇÃO.**
- IV – AS *THREADS* PERTENCEM A UM DETERMINADO PROCESSO.**

DAS AFIRMATIVAS ANTERIORES, QUAIS ESTÃO CORRETAS?

- A) II e III.**
- B) I e IV.**
- C) I, II e IV.**
- D) I, II.**

2. QUAL O VALOR IMPRESSO PELO SCRIPT A SEGUIR?:?

CODIGO 14 - SCRIPT PROCESSOS.PY

MINHA_LISTA = []

DEF ADICIONA():

FOR I IN RANGE(100):

MINHA_LISTA.APPEND(1)

IF __NAME__ == '__MAIN__':

TAREFAS = []

FOR INDICE IN RANGE(10):

T = THREAD(TARGET=ADICIONA)

TAREFAS.APPEND(T)

T.START()

FOR INDICE IN RANGE(10):

P = THREAD(TARGET=ADICIONA)

TAREFAS.APPEND(T)

P.START()

FOR TAREFA IN TAREFAS:

TAREFA.JOIN()

PRINT(LEN(MINHA_LISTA))

A) 100.

B) 1000.

C) 2000.

D) 10000.

GABARITO

1. Observe as afirmativas e responda:

I – É possível alcançar a concorrência em processadores de apenas um núcleo.

II – O paralelismo é uma técnica para permitir executar várias operações ao mesmo tempo.

III – Programas e processos têm a mesma definição.

IV – As *threads* pertencem a um determinado processo.

Das afirmativas anteriores, quais estão corretas?

A alternativa "C" está correta.

A única afirmação errada é a III, pois processo é um programa em execução.

2. Qual o valor impresso pelo script a seguir:?

Código 14 - Script processos.py

```
minha_lista = []

def adiciona():
    for i in range(100):
        minha_lista.append(1)

if __name__ == '__main__':
    tarefas = []

    for indice in range(10):
        t = Thread(target=adiciona)
        tarefas.append(t)
        t.start()

    for indice in range(10):
        p = Thread(target=adiciona)
        tarefas.append(p)
        p.start()

    for tarefa in tarefas:
        tarefa.join()

    print(len(minha_lista))
```

A alternativa "B" está correta.

Apenas a *thread* consegue acessar a variável global *minha_lista*. São executadas 10 *threads* X 100 iterações = 1000.

MÓDULO 3

- Identificar o Python como ferramenta para desenvolvimento web

INTRODUÇÃO

Atualmente, existem diversas opções de linguagem de programação para desenvolver aplicações *web*, sendo as principais: PHP, ASP.NET, Ruby e Java.

De acordo com o *site* de pesquisas W3Techs, o PHP é a linguagem mais utilizada nos servidores, com um total de 79% de todos os servidores utilizando essa linguagem.

Grande parte da utilização do PHP se deve aos gerenciadores de conteúdo, como WordPress, Joomla e Drupal, escritos em PHP e que tem muita representatividade na *web* hoje.

② VOCÊ SABIA?

O Python também pode ser utilizado para desenvolver aplicações *web*?

Atualmente existem diversos *frameworks* que facilitam a criação de aplicações *web* em Python.

Os *frameworks* podem ser divididos em, basicamente, dois tipos: *full-stack* e não *full-stack*.

FRAMEWORKS FULL-STACK

Os *frameworks full-stack* disponibilizam diversos recursos internamente, sem a necessidade de bibliotecas externas. Dentre os recursos, podemos citar:

Respostas à requisição;

Mecanismos de armazenamento (acesso a banco de dados);

Suporte a modelos (*templates*);

Manipulação de formulários;

Autenticação;

Testes;

Servidor para desenvolvimento, entre outros.

O principal *framework* Python *full-stack* é o Django.

FRAMEWORKS NÃO FULL-STACK

Estes oferecem os recursos básicos para uma aplicação *web*, como resposta a requisição e suporte a modelos. Os principais *frameworks* dessa categoria são o Flask e o CherryPy.

Normalmente, para aplicações maiores, são utilizados os *frameworks full-stack*, que também são um pouco mais complexos de se utilizar.

Para este módulo, usaremos o Flask.

CONCEITOS

O FLASK É UM MICRO *FRAMEWORK WEB* EM PYTHON.

Pela descrição de seus desenvolvedores, micro não quer dizer que toda a aplicação precisa ficar em apenas um arquivo ou que falta alguma funcionalidade, mas que o núcleo de funcionalidades dele é limpo e simples, porém extensível.

O desenvolvedor fica livre para escolher qual suporte a modelos (*templates*) utilizar, qual biblioteca de acesso ao banco de dados e como lidar com formulários. Existem inúmeras extensões compatíveis com o Flask.

INICIAR O DESENVOLVIMENTO COM FLASK É MUITO SIMPLES E MOSTRAREMOS ISSO AO LONGO DESTE MÓDULO. APESAR DE NÃO SER FULL-STACK, O FLASK DISPONIBILIZA UM

SERVIDOR WEB INTERNO PARA DESENVOLVIMENTO.

A configuração padrão inicia um servidor local na porta 5000, que pode ser acessado por:
`http://127.0.0.1:5000`.

Iremos começar com uma aplicação *web* básica que retorna “Olá, mundo.” quando acessamos pelo navegador o nosso servidor em `http://127.0.0.1:5000`.

O nome do nosso *script* é `flask1.py` e está apresentado no **Código 15** a seguir.

Código 15 - *Script flask1.py*

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route('/')
```

```
def ola_mundo():
```

```
    return "Olá, mundo."
```

```
if __name__ == '__main__':
```

```
    app.run()
```

Código 15 - Saída do *script flask1*

```
C:\Users\ftoli\PycharmProjects\estacio_ead\venv\Scripts\python.exe C:/Users/ftoli/Py
```

```
* Serving Flask app "principal" (lazy loading)
```

```
* Enviroment: production
```

```
WARNING: This is a development server. Do not use it in a production deployment.
```

Use a production WSGI server instead.

```
* Debug mode: off
```

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Na primeira linha, importamos a classe *flask*, que é a classe principal do *framework*. É a partir de uma instância dessa classe que criaremos nossa aplicação *web*. Na linha 2, é criada uma instância da classe *flask*, onde passamos `__name__` como argumento para que o Flask consiga localizar, na aplicação, arquivos estáticos, como *css* e *javascript*, e arquivos de modelos (*templates*), se for o caso.

② VOCÊ SABIA?

O Flask utiliza o conceito de rotas para direcionar o acesso às páginas (ou *endpoints*). As rotas servem para guiar as requisições feitas ao servidor para o trecho de código que deve ser executado. Os nomes das rotas são os nomes utilizados pelo usuário para navegar na sua aplicação.

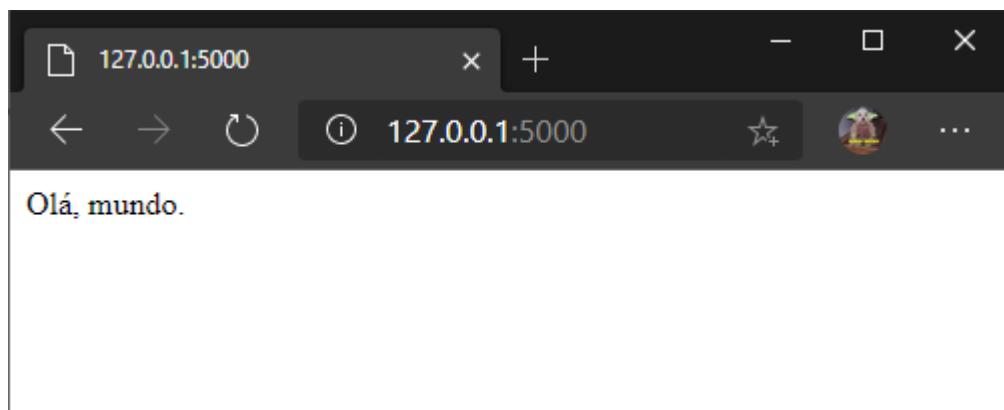
Por isso, utilize nomes com significado e de fácil memorização. No caso do Flask, as rotas são direcionadas para funções, que devem retornar algum conteúdo.

Na linha 5, utilizamos o decorador `@app.route('/')` para criar uma rota para a URL raiz da nossa aplicação ('/'). Esse decorador espera como parâmetro a rota, ou URL, que será utilizada no navegador, por exemplo.

Toda requisição feita para uma rota é encaminhada para a função imediatamente abaixo do decorador. No nosso caso, para a função `ola_mundo` (linha 6). O retorno dessa função é a resposta que o nosso servidor enviará ao usuário. Pela linha 7, a função `ola_mundo` retorna a string “Olá, mundo.”.

Ao iniciar a execução do *script*, recebemos no console algumas informações, incluindo em qual endereço o servidor está “escutando”, observe à direita do Codigo 16.

Ao digitar no navegador a URL <http://127.0.0.1:5000>, será exibida a frase “Olá, mundo.”, como na imagem da **figura 2**.



Fonte: Autor

⌚ Figura 2 - Resultado do acesso a <http://127.0.0.1:5000>.

APRIMORANDO ROTAS

Para ilustrar melhor o uso de rotas, vamos alterar o exemplo anterior de forma que a rota (URL) para a função `ola_mundo` seja `http://127.0.0.1:5000/ola`.

Observe o **Código 16** e compare com a anterior. Veja que o parâmetro do decorador `@app.route` da linha 5 agora é '`/ola`'.

Código 16 - Script `flask2.py`

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route('/ola')
```

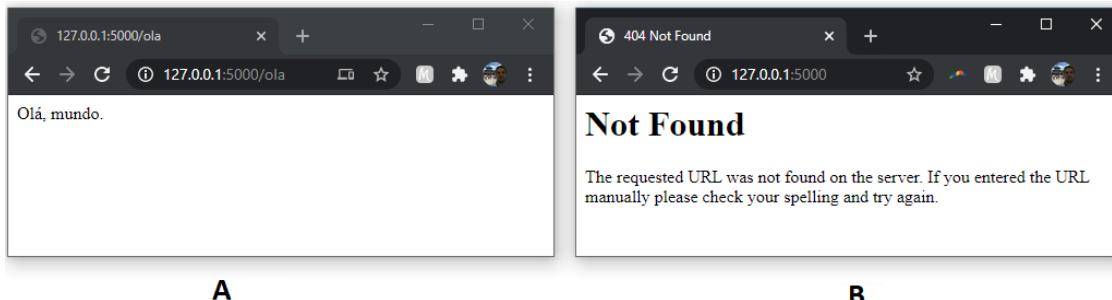
```
def ola_mundo():
```

```
    return "Olá, mundo."
```

```
if __name__ == '__main__':
```

```
    app.run()
```

Se acessarmos no navegador `http://127.0.0.1:5000/ola`, recebemos como resposta “Olá, mundo.” (**figura 3 A**), porém, ao acessar `http://127.0.0.1:5000`, recebemos um erro (404) de página não encontrada (*Not Found*), pois a rota para a URL raiz da aplicação não existe mais (**figura 3 B**).



Fonte: Autor

Figura 3 A - Resultado do acesso a `http://127.0.0.1:5000/ola`. Figura 3 B - Resultado do acesso a `http://127.0.0.1:5000`.

Vamos acertar nossa aplicação para criar, também, uma rota para a URL raiz da aplicação (`@app.route('/')`).

Vamos chamar a função dessa rota de *index* e criar essa rota conforme linhas 5 a 7. Veja o resultado no **Código 17** a seguir.

Código 17 - Script `flask3.py`

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route('/')
```

```
def index():
```

```
    return "Página principal."
```

```
@app.route('/ola')
```

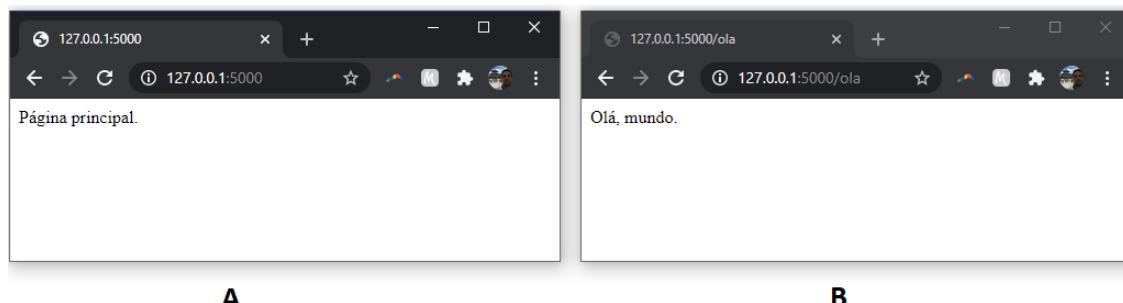
```
def ola_mundo():
```

```
    return "Olá, mundo."
```

```
if __name__ == '__main__':
```

```
    app.run()
```

Veja, na imagem a seguir, que agora podemos acessar tanto a rota para função *índice* (URL: /) (**figura 4 A**), onde é retornado “Página principal”, quanto a rota da função *ola_mundo* (URL: /ola) (**figura 4 B**), que é retornado “Olá, mundo.”.



Fonte: Autor

- ▣ Figura 4 A - Resultado do acesso a <http://127.0.0.1:5000>. Figura 4 B - Resultado do acesso a <http://127.0.0.1:5000/ola>.

RECEBENDO PARÂMETROS

O decorador de rota (*route*) também permite que sejam passados parâmetros para as funções. Para isso, devemos colocar o nome do parâmetro da função entre < e > na URL da rota.

No exemplo do **Código 18** a seguir, vamos mudar a rota da função *ola_mundo* de forma que seja possível capturar e retornar o nome passado como parâmetro.

Codigo 18 - Script flask4.py

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route('/')
```

```
def index():
```

```
    return "Página principal."
```

```
@app.route('/ola/<nome>')
```

```
def ola_mundo(nome):
```

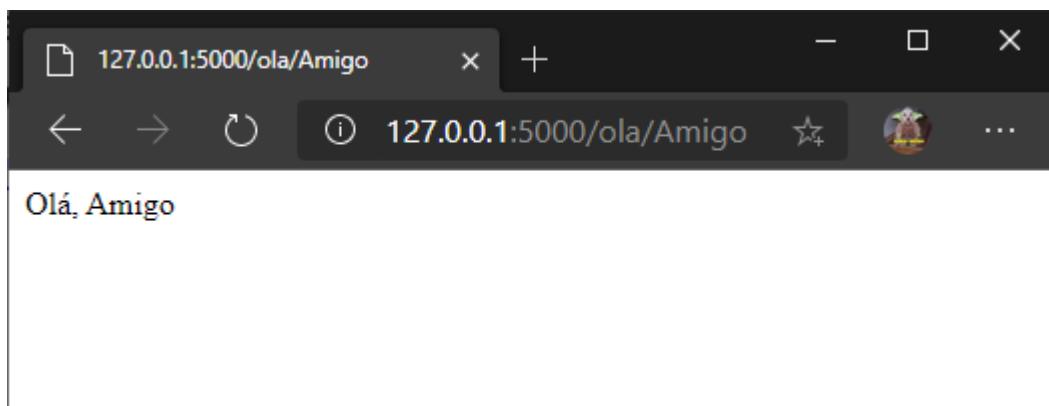
```
    return "Olá, " + nome
```

```
if __name__ == '__main__':
```

```
    app.run()
```

Observe a URL da rota na linha 9, em que utilizamos a variável *nome* entre < e >, que é o mesmo nome do parâmetro da função *ola_mundo*. Isso indica que qualquer valor que for adicionado à URL após '/ola/' será passado como argumento para a variável *nome* da função *ola_mundo*.

Veja na **figura 5** a seguir, em que acessamos a URL <http://127.0.0.1:5000/ola/Amigo>.



Fonte: Autor

Figura 5 - Resultado do acesso a <http://127.0.0.1:5000/ola/Amigo>.

Porém, se tentarmos acessar a URL <http://127.0.0.1:5000/ola>, receberemos um erro, pois removemos a rota para essa URL. Para corrigir esse problema, vamos adicionar uma segunda rota para a mesma função, bastando adicionar outro decorador `@app.route` para a mesma função *ola_mundo*, conforme **Codigo 19** a seguir.

Código 19 - Script flask5.py

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route('/')
```

```
def index():
```

```
    return "Página principal."
```

```
@app.route('/ola/')
```

```
@app.route('/ola/<nome>')
```

```
def ola_mundo(nome="mundo"):
```

```
    return "Olá, " + nome
```

```
if __name__ == '__main__':
```

```
    app.run()
```

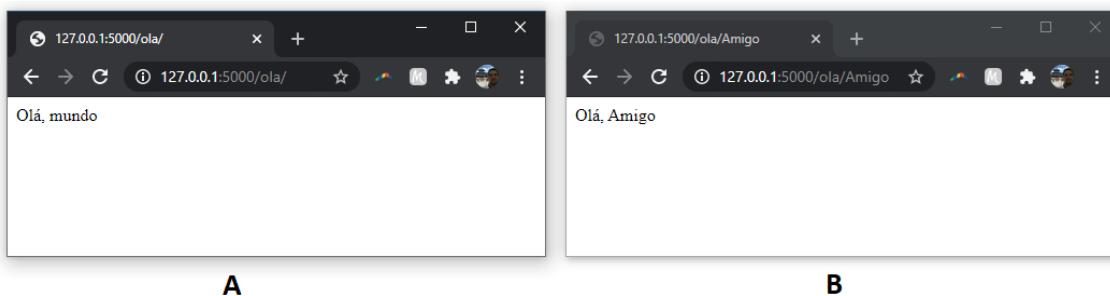
Observe que agora temos duas rotas para a mesma função (linhas 9 e 10). Em qualquer uma das URLs, o usuário será direcionado para a função *ola_mundo*.

► ATENÇÃO

OBS: A rota com URL '/ola/' aceita requisições tanto para '/ola' quanto '/ola/'.

Como nessa nova configuração, o parâmetro *nome* pode ser vazio quando acessado pela URL '/ola', definimos o valor padrão “*mundo*” para ele (linha 11).

Ao acessar essas duas URLs '/ola/' e '/ola/Amigo', obtemos os resultados exibidos nos resultados exibidos na **figura 6 A e 6 B**, respectivamente.



Fonte: Autor

- ▣ Figura 6 A - Resultado do acesso a <http://127.0.0.1:5000/ola/>. Figura 6 B - Resultado do acesso a <http://127.0.0.1:5000/ola/Amigo>.

MÉTODOS HTTP

AS APLICAÇÕES WEB DISPONIBILIZAM DIVERSOS MÉTODOS PARA ACESSAR UMA URL: GET, POST, PUT E DELETE. POR PADRÃO, AS ROTAS DO FLASK SOMENTE RESPONDEM ÀS REQUISIÇÕES GET. PARA RESPONDER A OUTROS MÉTODOS, É NECESSÁRIO EXPLICITAR, NA ROTA, QUAIS MÉTODOS SERÃO ACEITOS.

Para isso, precisamos utilizar o parâmetro *methods* do decorador `@app.route()`, que espera uma lista de *strings* com o nome dos métodos aceitos.

Observe no **Código 20**, a seguir, em que criamos a rota para a função *logar* e passamos como argumento uma lista contendo duas *strings*, 'GET' e 'POST' (linha 15). Isso indica que essa rota deve responder às requisições do tipo GET e POST.

Código 20 - Script flask6.py

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
app.debug = True
```

```
@app.route('/')
```

```
def index():
```

```
    return "Página principal."
```

```
@app.route('/ola/')
```

```
@app.route('/ola/<nome>')
```

```
def ola_mundo(nome):
```

```
    return "Olá, " + nome
```

```
@app.route('/logar', methods=['GET', 'POST'])
```

```
def logar():
```

```
    if request.method == 'POST':
```

```
        return "Recebeu post! Fazer login!"
```

```
    else:
```

```
        return "Recebeu get! Exibir FORM de login."
```

```
if __name__ == '__main__':
```

```
    app.run()
```

Para verificar o método que foi utilizado na requisição, usamos o atributo *method* do objeto *request*, que retorna uma das *strings*: GET, POST, PUT ou DELETE.

O objeto *request* é uma variável global disponibilizada pelo Flask e pode ser utilizada em todas as funções. Para cada requisição, um novo objeto *request* é criado e disponibilizado.

Com o objeto *request*, temos acesso a muitas outras propriedades da requisição, como: *cookie*, parâmetros, dados de formulário, *mimetype* etc.

Neste exemplo, utilizamos o atributo *method* do objeto *request* para verificar o método passado na requisição e retornar conteúdos diferentes dependendo do método (linhas 17 a 20).

 ATENÇÃO

Caso seja requisitada uma rota que exista, porém o método não seja aceito pela rota, o erro retornado é o 405 - *Method Not Allowed* (método não permitido), e não o 404 – *Not Found* (não encontrado), como ocorre quando a rota não existe.

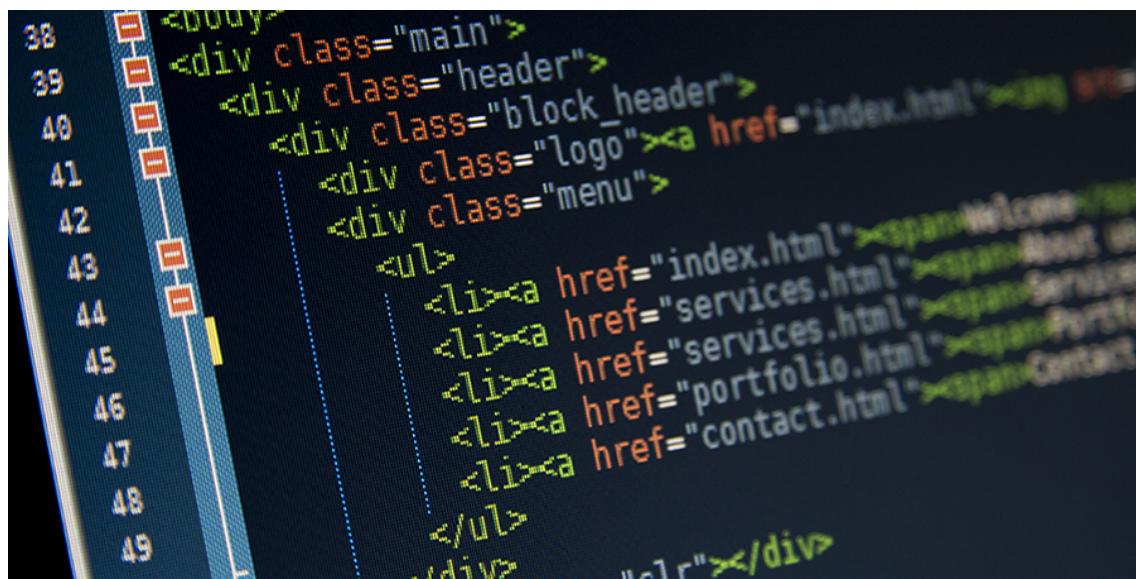
Observe que ativamos o modo *debug* do servidor interno do Flask (linha 5). Isso nos permite visualizar melhor os avisos e erros durante o desenvolvimento.

UTILIZANDO MODELOS

As funções no **Flask** precisam retornar algo. O retorno das funções pode ser uma *string* simples e até uma *string* contendo uma página inteira em HTML. Porém, criar páginas dessa maneira é muito complicado, principalmente devido à necessidade de escapar (*scape*) o HTML, para evitar problemas de segurança, por exemplo.

Para resolver esse problema, o Flask, por meio da extensão *Jinja2*, permite utilizar modelos (*templates*) que são arquivos texto com alguns recursos a mais, inclusive escape automático.

No caso de aplicações *web*, os modelos normalmente são páginas HTML, que são pré-processadas antes de retornarem ao requisitante, abrindo um novo leque de possibilidades no desenvolvimento *web*.



```
38 <div>
39   <div class="main">
40     <div class="header">
41       <div class="block_header">
42         <div class="logo"><a href="index.html">Index</a></div>
43         <div class="menu">
44           <ul>
45             <li><a href="index.html">Welcome</a></li>
46             <li><a href="services.html">About Us</a></li>
47             <li><a href="services.html">Services</a></li>
48             <li><a href="portfolio.html">Portfolio</a></li>
49             <li><a href="contact.html">Contact</a></li>
50           </ul>
51         </div>
52       </div>
53     </div>
54   </div>
55 </div>
```

Fonte: Melody Smart/Shutterstock

A cada requisição, podemos alterar uma mesma página de acordo com um contexto (valores de variáveis, por exemplo). Com a utilização de marcadores (*tags*) especiais, é possível injetar valores de variáveis no corpo do HTML, criar laços (*for*), condicionantes (*if/else*), filtros etc.

1

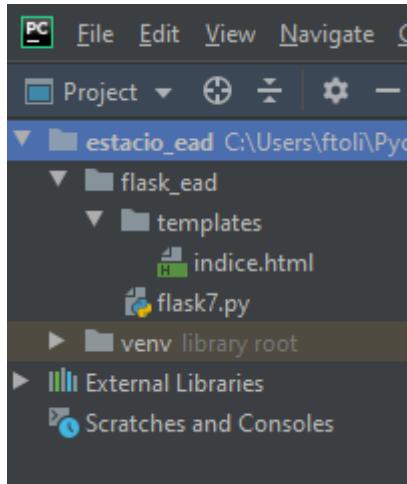
Para criar e utilizar os modelos, por convenção, os HTMLs precisam ficar dentro da pasta *templates*, no mesmo nível do arquivo que contém a aplicação Flask.

2

Para ilustrar, no próximo exemplo (**Código 21**), vamos alterar nossa aplicação de forma que seja retornada uma página HTML ao se acessar a raiz da aplicação ('/').

3

Para isso, vamos criar um arquivo chamado *indice.html* na pasta *templates*, no mesmo nível do script *flask7.py*, conforme vemos na árvore de diretório abaixo.



Fonte: MAutor

O conteúdo do modelo *indice.html* está exibido abaixo do **Código 21**

4

Para o modelo ser acessado na URL raiz ('/'), a função *index()* deve ser a responsável por retorná-lo. Para isso, vamos utilizar a função *render_template* disponibilizada pelo Flask, que recebe o nome do arquivo que desejamos retornar.

Observe a linha 7 do script *flask7.py*, em que utilizamos essa função e passamos “*indice.html*” como parâmetro.

Código 21 - Script *flask7.py*

```
from flask import Flask, render_template
```

```
app = Flask(__name__)

@app.route('/')
def index():
    return render_template('indice.html')

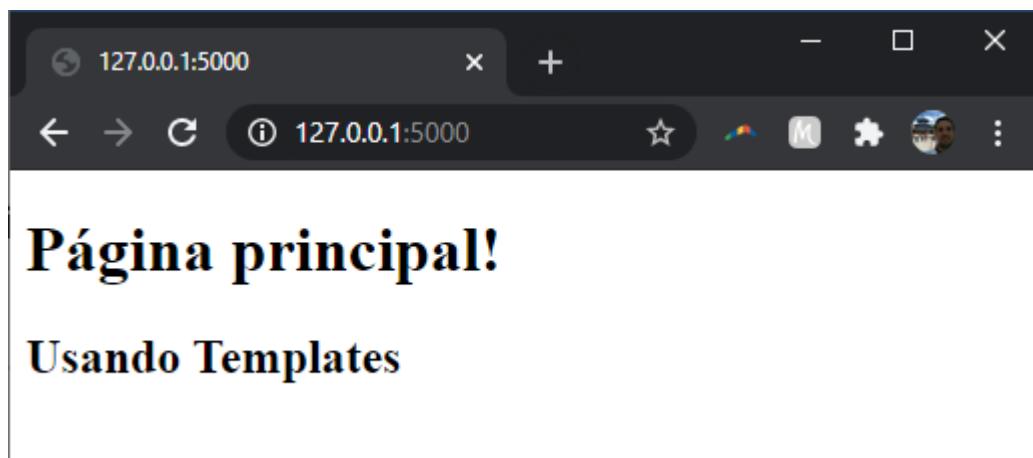
@app.route('/ola/')
@app.route('/ola/<nome>')
def ola_mundo(nome="mundo"):
    return "Olá, " + nome

if __name__ == '__main__':
    app.run()
```

Código 21 - Modelo indice.html

```
<!DOCTYPE html>
<html>
<body>
<h1>Página principal!</h1>
<h2>Usando templates</h2>
</body>
</html>
```

Ao acessar <http://127.0.0.1:5000>, recebemos o resultado exibido na **figura 7**.



Fonte: Autor

Figura 7 - Resultado do acesso a <http://127.0.0.1:5000>.

Como dito anteriormente, os modelos são páginas HTML turbinadas, nas quais podemos utilizar delimitadores especiais para alterar nossa página. Os dois tipos de delimitadores são:

Expressões: {{ ... }}

Que serve para escrever algo no modelo, como o valor de uma variável.

Declarações: {% ... %}

Utilizado em laços e condicionantes, por exemplo.

Antes de serem retornadas ao usuário, essas páginas são renderizadas, ou seja, os delimitadores são computados e substituídos.

No próximo exemplo, **Código 22**, vamos alterar a nossa aplicação de forma que o nome passado como parâmetro para a rota '/ola/<nome>' seja exibido dentro do HTML.

Para isso, vamos criar um arquivo chamado *ola.html* na pasta *templates*, conforme abaixo no **Código 22**. Observe que utilizamos o delimitador de expressões com a variável ({{nome_recebido}}), indicando que vamos escrever o conteúdo dessa variável nesse local após a renderização (linha 4 do arquivo *ola.html*).

Código 22 - Script flask8.py

```
from flask import Flask, render_template
```

```
app = Flask(__name__)
```

```
@app.route('/')
```

```
def index():
```

```
    return render_template('indice.html')
```

```
@app.route('/ola/')
```

```
@app.route('/ola/<nome>')
```

```
def ola_mundo(nome="mundo"):
```

```
    return "Olá, ", nome
```

```
if __name__ == '__main__':
```

```
    app.run()
```

Código 22 - Modelo *ola.html*

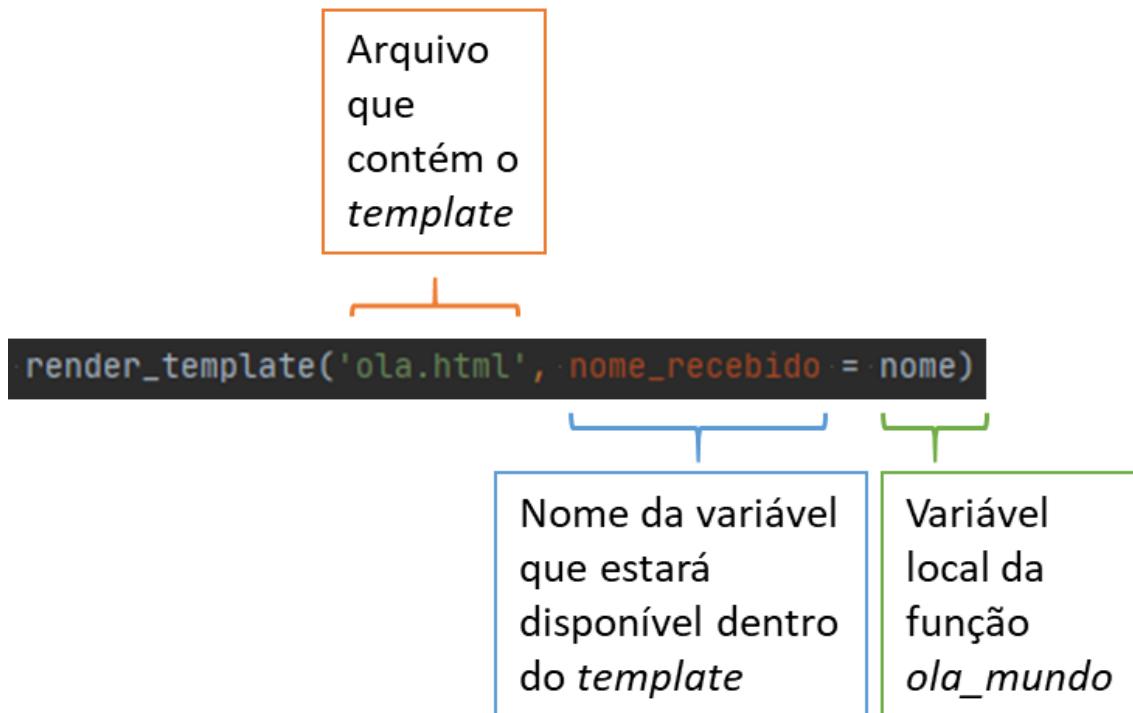
```

<!DOCTYPE html>
<html>
<body>
<h1>Olá, {{ nome_recebido }}</h1>
</body>
</html>

```

Além de informar a página que queremos renderizar na função ***render_template***, podemos passar uma ou mais variáveis para essa função. Essas variáveis ficarão disponíveis para serem utilizadas em expressões (`{{ }}`) e declarações (`{% %}`) dentro do modelo (*template*).

No exemplo, desejamos exibir o *nome* passado via URL na página. Para passar a variável ***nome*** para o HTML, precisamos chamar a função *render_template* com um parâmetro a mais, conforme linha 12 e destacado a seguir (**figura 8**).



Fonte: Autor

- ▣ Figura 8 - Destaque da sintaxe da função *render_template* do Flask.

A variável *nome_recebido* estará disponível para ser usada dentro do modelo *ola.html*. Esse nome pode ser qualquer outro escolhido pelo programador, mas lembre-se de que ele será usado, também, dentro do modelo.

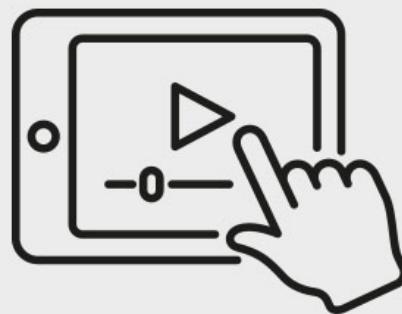
Para finalizar, utilizamos o delimitador de expressões envolvendo o nome da variável (linha 4 do html). Ao ser renderizada, essa expressão será substituída pelo valor da variável, conforme **figura 9** a seguir, onde passamos a *string* “Amigo” para a variável *nome* e, consequentemente, *nome_recebido*.



Fonte: Autor

- Figura 9 - Resultado do acesso a <http://127.0.0.1:5000/ola/Amigo>.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. CONSIDERE O CÓDIGO A SEGUIR, EM QUE TEMOS UM SERVIDOR FLASK ESCUTANDO NA PORTA 5000, E RESPONDA:

EXERCICIO1.PY

FROM FLASK IMPORT FLASK

APP = FLASK(__NAME__)

@APP.ROUTE('/OLA')

```
DEF OLA_MUNDO():
    RETURN "OLÁ, MUNDO"
```

```
@APP.ROUTE('/OLA')
DEF OLA_MUNDO(NOME="MUNDO"):
    RETURN "OLÁ, " + NOME
```

```
IF __NAME__ == '__MAIN__':
    APP.RUN()
```

O QUE SERÁ APRESENTADO NO NAVEGADOR SE ACESSARMOS A URL
HTTP://127.0.0.1:5000/OLA/EAD?

- A) Olá, mundo.
- B) Olá, mundo.
- C) Olá, EAD.
- D) O programa vai apresentar um erro.

2. CONSIDERE O CÓDIGO A SEGUIR, NO QUAL TEMOS UM SERVIDOR FLASK ESCUTANDO NA PORTA 5000, E RESPONDA:

EXERCICIO2.PY

```
FROM FLASK IMPORT FLASK
```

```
APP = FLASK(__NAME__)
```

```
@APP.ROUTE('/OLA', METHODS=['POST'])
DEF OLA_POST():
    RETURN "OLÁ, GET"
```

```
@APP.ROUTE('/OLA')

DEF OLA_GET(NOME="MUNDO"):
    RETURN "OLÁ, POST"

IF __NAME__ == '__MAIN__':
    APP.RUN()
```

**O QUE SERÁ APRESENTADO NO NAVEGADOR SE ACESSARMOS A URL
HTTP://127.0.0.1:5000/OLA?**

- A) Olá, GET.
- B) Olá, GET.
Olá, POST.
- C) Olá, POST.
- D) O programa vai apresentar um erro.

GABARITO

1. Considere o código a seguir, em que temos um servidor Flask escutando na porta 5000, e responda:

exercicio1.py

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route('/ola')
```

```
def ola_mundo():
```

```
    return "Olá, mundo"
```

```
@app.route('/ola')
```

```
def ola_mundo(nome="mundo"):
```

```
    return "Olá, " + nome
```

```
if __name__ == '__main__':
    app.run()
```

O que será apresentado no navegador se acessarmos a URL
<http://127.0.0.1:5000/ola/EAD>?

A alternativa "C" está correta.

A URL acessada está de acordo com a rota da linha 9.

2. Considere o código a seguir, no qual temos um servidor Flask escutando na porta 5000, e responda:

exercicio2.py

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route('/ola', methods=['POST'])
def ola_post():
    return "Olá, GET"
```

```
@app.route('/ola')
def ola_get(nome="mundo"):
    return "Olá, POST"
```

```
if __name__ == '__main__':
    app.run()
```

O que será apresentado no navegador se acessarmos a URL <http://127.0.0.1:5000/ola>?

A alternativa "C" está correta.

O navegador utiliza, por *default*, o método GET. Com isso, será executada a rota para a função *ola_get*, da linha 10.

MÓDULO 4

- Identificar o Python como ferramenta para ciência de dados

INTRODUÇÃO

Desde o século XVII, as ciências experimentais e teóricas são reconhecidas pelos cientistas como os paradigmas básicos de pesquisa para entender a natureza. De umas décadas para cá, a simulação computacional de fenômenos complexos evoluiu, criando o terceiro paradigma, a ciência computacional.

A ciência computacional fornece ferramentas necessárias para tornar possível a exploração de domínios inacessíveis à teoria ou experimento.

Com o aumento das simulações e experimentos, mais dados são gerados e um quarto paradigma emerge, que são as tecnologias e técnicas associadas à ciência de dados.

A ciência de dados é uma área de conhecimento que envolve a utilização de dados para gerar impactos em uma instituição, seja uma universidade, uma empresa, um órgão federal etc., de forma a resolver um problema real utilizando os dados.



Fonte: metamorworks/Shutterstock

Em 1996, Fayyad (1996) apresentou a definição clássica do processo de descoberta de conhecimento em bases de dados, conhecido por KDD (*Knowledge Discovery in Databases*):

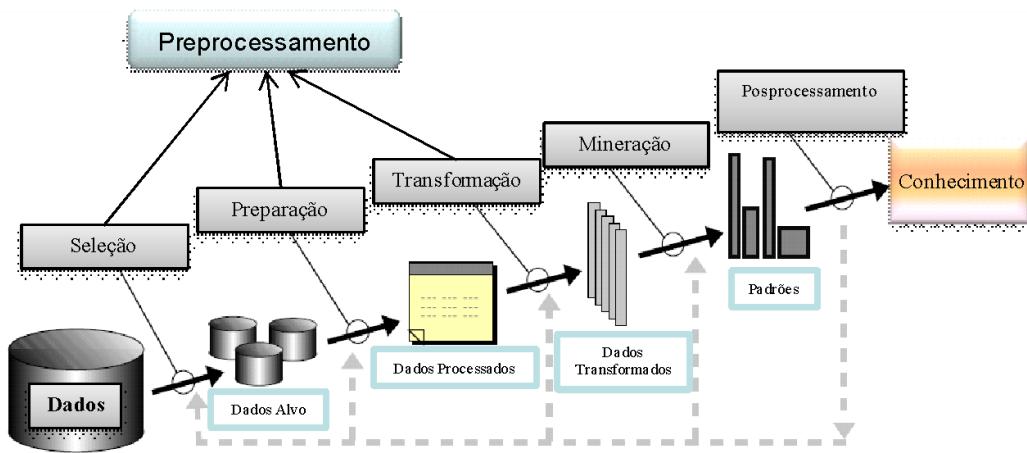
“KDD É UM PROCESSO, DE VÁRIAS ETAPAS, NÃO TRIVIAL, INTERATIVO E ITERATIVO, PARA IDENTIFICAÇÃO DE PADRÕES COMPREENSÍVEIS, VÁLIDOS, NOVOS E POTENCIALMENTE ÚTEIS A PARTIR DE GRANDES CONJUNTOS DE DADOS”.

As técnicas de KDD (FAYYAD, 1996), também conhecidas como mineração de dados, normalmente se referem à extração de informações implícitas, porém úteis, de uma base de dados.

Essas aplicações, tipicamente, envolvem o uso de mineração de dados para descobrir um novo modelo, e então os analistas utilizam esse modelo em suas aplicações.

O processo de KDD é basicamente composto por três grandes etapas: **pré-processamento, mineração de dados e pós-processamento**.

A **figura 10**, a seguir, mostra todo o processo de KDD.



Fonte: FAYYAD, 1996

Figura 10 - Visão geral dos passos que compõe o processo de KDD

A primeira etapa do processo de KDD, conhecida como pré-processamento, é responsável por selecionar, preparar e transformar os dados que serão utilizados pelos algoritmos de mineração.

Algumas atividades envolvidas no reprocessamento são:

COLETA E INTEGRAÇÃO:

Quando é necessário que dados provenientes de diversas fontes sejam consolidados em uma única base de dados. Esta atividade é bastante encontrada na construção de data *warehouses*;

CODIFICAÇÃO:

Significa transformar a natureza dos valores de um atributo. Isto pode acontecer de duas diferentes formas: uma transformação de dados numéricos em categóricos — codificação numérico-categórica, ou o inverso — codificação categórico-numérica;

CONSTRUÇÃO DE ATRIBUTOS:

Após a coleta e integração dos dados, pode ser necessário criar colunas em uma tabela, por exemplo, refletindo alguma transformação dos dados existentes em outras colunas;

LIMPEZA DOS DADOS:

Pode ser subdividida em complementação de dados ausentes, detecção de ruídos, e eliminação de dados inconsistentes;

A PARTIÇÃO DOS DADOS:

Consiste em separar os dados em dois conjuntos disjuntos. Um para treinamento (abstração do modelo de conhecimento) e outro para testes (avaliação do modelo gerado).

A segunda etapa do KDD, conhecida como mineração de dados, é a aplicação de um algoritmo específico para extrair padrões de dados. Hand (2001) define a etapa de mineração de dados da seguinte forma:

“MINERAÇÃO DE DADOS É A ANÁLISE DE (QUASE SEMPRE GRANDES) CONJUNTOS DE DADOS OBSERVADOS PARA DESCOBRIR RELAÇÕES ESCONDIDAS E PARA CONSOLIDAR OS DADOS DE UMA FORMA TAL QUE ELES SEJAM INTELIGÍVEIS E ÚTEIS AOS SEUS DONOS.”

Esta etapa, normalmente, é a que atrai maior atenção, por ser ela que revela os padrões ocultos nos dados.

OS ALGORITMOS DE MINERAÇÃO PODEM SER CLASSIFICADOS COMO SUPERVISIONADOS E NÃO SUPERVISIONADOS. NOS PRIMEIROS, OS ALGORITMOS “APRENDEM” BASEADOS NOS VALORES QUE CADA DADO JÁ POSSUI. OS ALGORITMOS SÃO TREINADOS (AJUSTADOS),

APLICANDO UMA FUNÇÃO E COMPARANDO O RESULTADO COM OS VALORES EXISTENTES.

Já nos não supervisionados, os dados não foram classificados previamente e os algoritmos tentam extrair algum padrão por si só.

A seguir, serão apresentados alguns algoritmos que podem ser realizados durante a etapa de mineração de dados.

Não supervisionados:

Regras de associação

Uma das técnicas de mineração de dados mais utilizada para comércio eletrônico, cujo objetivo é encontrar regras para produtos comprados em uma mesma transação. Ou seja, a presença de um produto em um conjunto implica a presença de outros produtos de outro conjunto; com isso, *sites* de compras nos enviam sugestões de compras adicionais, baseado no que estamos comprando.

Agrupamento

Reúne, em um mesmo grupo, objetos de uma coleção que mantenham algum grau de afinidade. É utilizada uma função para maximizar a similaridade de objetos do mesmo grupo e minimizar entre elementos de outros grupos.

Supervisionados:

Classificação

Tem como objetivo descobrir uma função capaz de mapear (classificar) um item em uma das várias classes predefinidas. Se conseguirmos obter a função que realiza esse mapeamento, qualquer nova ocorrência pode ser também mapeada, sem a necessidade de conhecimento prévio da sua classe;

Regressão linear

É uma técnica para se estimar uma variável a partir de uma função. A regressão, normalmente, tem o objetivo de encontrar um valor que não foi computado inicialmente.

- **Atenção!** Para visualização completa da tabela utilize a rolagem horizontal
- A última etapa do KDD, o pós-processamento, tem como objetivo transformar os padrões dos dados obtidos na etapa anterior, de forma a torná-los inteligíveis, tanto ao analista de dados quanto ao especialista do domínio da aplicação (SOARES, 2007).

CONCEITOS

Vamos apresentar algumas situações de forma a explicar alguns algoritmos de mineração e como eles podem ser implementados em Python.

Utilizaremos a biblioteca Pandas para realizar a leitura de dados, a biblioteca Scikit-Learn para realizar o treinamento e utilização dos algoritmos de mineração de dados e a biblioteca *matplotlib* para gerar a visualização de resultados.

SUPERVISIONADO – REGRESSÃO LINEAR

Neste exemplo, vamos utilizar uma série histórica fictícia de casos de dengue de uma determinada cidade e, com o auxílio do algoritmo supervisionado de **regressão linear**, predizer casos futuros.

A série está em uma planilha (arquivo CSV) com duas colunas, **ano** e **casos** (número de casos). Na planilha, temos o número de casos de 2001 a 2017. Vamos utilizar essa série histórica e aplicar o algoritmo de regressão linear para estimar os casos de dengue para o ano de 2018.

No **Código 23**, a seguir, temos o código do *script regressao.py*, o arquivo CSV (dados_dengue.csv) e a saída do console.

Código 23 - *Script regressao.py*

```
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
import pandas
```

```
##### Pré-processamento #####
```

```

# Coleta e Integração
arquivo = pandas.read_csv('dados_dengue.csv')

anos = arquivo[['ano']]
casos = arquivo[['casos']]

##### Mineração #####
regr = LinearRegression()
regr.fit(X=anos, y=casos)

ano_futuro = [[2018]]
casos_2018 = regr.predict(ano_futuro)

print('Casos previstos para 2018 ->', int(casos_2018))

##### Pós-processamento #####
plt.scatter(anos, casos, color='black')
plt.scatter(ano_futuro, casos_2018, color='red')
plt.plot(anos, regr.predict(anos), color='blue')

plt.xlabel('Anos')
plt.ylabel('Casos de dengue')
plt.xticks([2018])
plt.yticks([int(casos_2018)])

plt.show()

```

Código 23 - Dados_dengue.csv

ano	casos
2017	450
2016	538
2015	269
2014	56
2013	165
2012	27
2011	156

2010,102
2009,86
2008,42
2007,79
2006,65
2005,58
2004,39
2003,23
2002,15
2001,28

Código 23 - Saída do script regressao.py

C:\Users\fotoli\PycharmProjects\estac

Casos previstos para 2018 -> 330

Após importar os módulos necessários, vamos passar para a primeira etapa do KDD, o pré-processamento. Neste caso simples, vamos realizar apenas a **coleta e integração** que, na prática, é carregar a planilha dentro do programa.

Para isso, utilizamos a função *read_csv* da biblioteca Pandas, passando como parâmetro o nome do arquivo (linha 7).

A CLASSE DA BIBLIOTECA **SCIKIT-LEARN** UTILIZADA PARA REALIZAR A REGRESSÃO LINEAR SE CHAMA **LINEARREGRESSION**. PARA REALIZAR A REGRESSÃO, ELA PRECISA DOS DADOS DE TREINAMENTO (PARÂMETRO X) E SEUS RESPECTIVOS RESULTADOS (PARÂMETRO Y).

No nosso exemplo, como desejamos estimar o número de casos, vamos utilizar os **anos** como dado de treinamento e o número de **casos** como resultado. Ou seja, teremos os **anos** no parâmetro X e os **casos** no parâmetro y.

Como estamos usando apenas uma variável para o parâmetro X, o ano, temos uma regressão linear simples e o resultado esperado é uma reta, onde temos os **anos** no eixo x e os **casos** no eixo y.

Após carregar o arquivo, vamos separar os dados das colunas nas respectivas variáveis.

Observe a sintaxe para obter os **anos** (linha 9) e **casos** (linha 10). O Pandas detecta, automaticamente, o nome das colunas e permite extrair os elementos das colunas utilizando o nome.

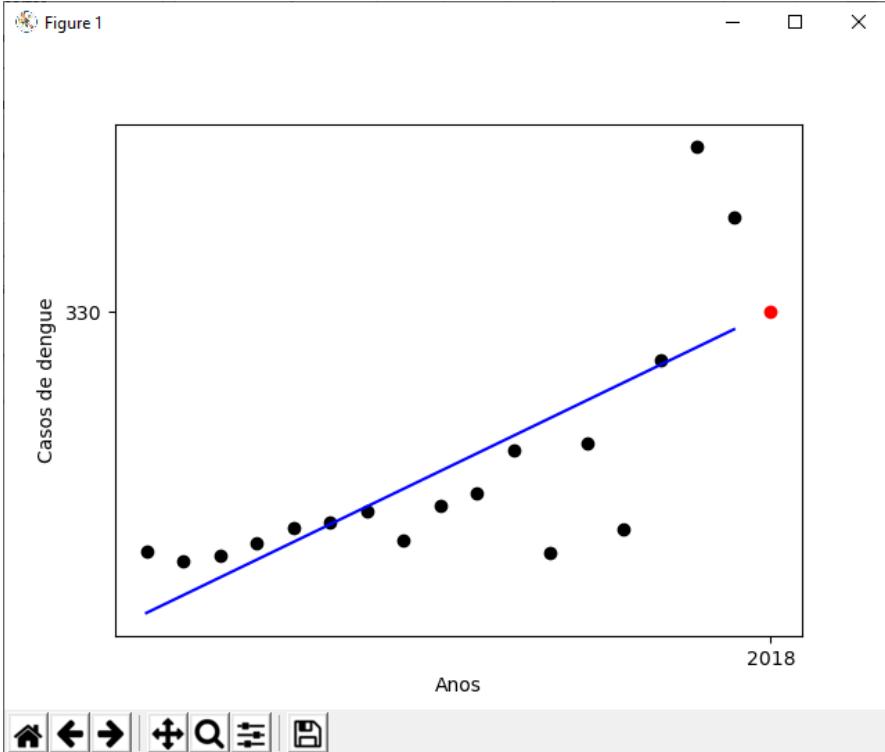
ATENÇÃO

O próximo passo é criar o objeto do tipo *LinearRegression* e atribuí-lo a uma variável (linha 13). Esse objeto será utilizado para treinar (ajustar) a equação da reta que será gerada pela regressão. Para realizar o treinamento (*fit*), precisamos passar os parâmetros X e y (linha 14).

Após a execução do método *fit*, o objeto *regr* está pronto para ser utilizado para predizer os casos para os anos futuros, utilizando o método *predict* (linha 17).

Ao chamar o método *predict* passando o ano 2018 como argumento, recebemos como retorno o número casos previsto para 2018, conforme impresso no console da **figura 33** (330).

A partir da linha 21, temos a etapa de pós-processamento, na qual utilizamos a biblioteca *matplotlib* para exibir um gráfico com os dados da série (pontos em preto), a reta obtida pela regressão, em azul, e o valor predito para o ano de 2018, em vermelho (**figura 11**).



Fonte: Autor

Figura 11 - Gráfico da série dos casos de dengue

SUPERVISIONADO – CLASSIFICAÇÃO

Os algoritmos de classificação são do tipo supervisionado, nos quais passamos um conjunto de características sobre um determinado item de uma classe de forma que o algoritmo consiga compreender, utilizando apenas as características, qual a classe de um item não mapeado.

Para este exemplo, vamos utilizar um conjunto de dados (*dataset*) criado em 1938 e utilizado até hoje: o *dataset* da flor de íris (*Iris Dataset*). Ele contém informações de cinquenta amostras de três diferentes classes de Flor de Íris (*Iris setosa*, *Iris virginica* e *Iris versicolor*).

NO TOTAL, SÃO QUATRO CARACTERÍSTICAS PARA CADA AMOSTRA, SENDO ELAS O COMPRIMENTO E A LARGURA, EM CENTÍMETROS, DAS SÉPALAS E PÉTALAS DE ROSAS.

Por ser um *dataset* muito utilizado e pequeno, o Scikit-Learn já o disponibiliza internamente.

Vamos treinar dois algoritmos de classificação, árvore de decisão e máquina de vetor suporte (*support vector machine* – SVM) para montar dois classificadores de flores de íris. A forma como são implementados esses algoritmos estão fora do escopo deste módulo.

Confira o *script* a seguir, **Codigo 24**, em que utilizamos esses dois algoritmos.

Codigo 24 - *Script* classificacao.py

```
from sklearn.datasets import load_iris, fetch_kddcup99
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, export_text, plot_tree
from sklearn.svm import SVC

##### Pré-processamento #####
# Coleta e Integração
iris = load_iris()

características = iris.data
rotulos = iris.target

print("Características:\n", características[:2])
print("Rótulos:\n", rotulos[:2])
print('#####')

# Partição dos dados
carac_treino, carac_teste, rot_treino, rot_teste = train_test_split(características, rotulos)

##### Mineração #####
#####----- Árvore de Decisão -----#####
arvore = DecisionTreeClassifier(max_depth=2)
arvore.fit(X=carac_treino, y=rot_treino)

rot_preditos = arvore.predict(carac_teste)
acuracia_arvore = accuracy_score(rot_teste, rot_preditos)
```

```
##### Máquina de Vetor Suporte #####
```

```
clf = SVC()
```

```
clf.fit(X=carac_treino, y=rot_treino)
```

```
rot_preditos_svm = clf.predict(carac_teste)
```

```
acuracia_svm = accuracy_score(rot_teste, rot_preditos_svm)
```

```
##### Pós-processamento #####
```

```
print("Acurácia Árvore de Decisão:", round(acuracia_arvore, 5))
```

```
print("Acurácia SVM:", round(acuracia_svm, 5))
```

```
print("#####")
```

```
r = export_text(arvore, feature_names=iris['feature_names'])
```

```
print("Estrutura da árvore")
```

```
print(r)
```

Codigo 24 - Saída do *script* classificacao.py

```
C:\Users\fotoli\PycharmProjects\estacio_ead\
```

Características:

```
[[5.1 3.5 1.4 0.2]
```

```
[4.9 3. 1.4 0.2]]
```

Rótulos:

```
[0 0]
```

```
#####
```

```
Acurácia Árvore de Decisão: 0.92105
```

```
Acurácia SVM: 0.97368
```

```
#####
```

Estrutura da árvore

```
|--- petal width (cm) <= 0.80
```

```
| |--- class: 0
```

```
|--- petal width (cm) > 0.80
```

```
| |--- petal width (cm) <= 1.75
```

```
| | |--- class: 1
```

```
|--- petal width (cm) > 1.75
```

```
| | |--- class: 2
```

```
Process finished with exit code 0
```

ETAPA 01

Na etapa de pré-processamento, vamos começar pela **coleta e integração**, que é a obtenção do *dataset* de flores utilizando a função *load_iris()* (linha 9). Esta função retorna um objeto onde podemos acessar as *características* das flores pelo atributo *data* e os *rótulos*, ou *classes* das flores, pelo atributo *target*.

ETAPA 02

Na linha 11, separamos as características das flores na variável *características*. Ela contém uma lista com 150 itens, onde cada item contém outra lista com quatro elementos. Observe o conteúdo dos dois primeiros itens desta lista no console.

Cada um dos quatro elementos corresponde ao comprimento da sépala, largura da sépala, comprimento da pétala e largura da pétala, respectivamente.

ETAPA 03

Na linha 12, separamos os rótulos (ou classes) na variável *rótulo*. Ela contém uma lista com 150 itens que variam entre 0, 1 ou 2. Cada número corresponde a uma classe de flor (0: Iris-Setosa; 1:Iris-Versicolour; 2:Iris-Virginica). Como dito na introdução deste módulo, esse mapeamento entre categorias e números se chama **codificação categórico-numérica**. Essa etapa de pré-processamento já foi realizada e disponibilizada pela função *load_iris()*.

ETAPA 03

Outra etapa de pré-processamento que precisaremos realizar é a **partição dos dados**. Ela nos permitirá verificar a qualidade do algoritmo de classificação. Para isso, precisamos particionar nossos dados em treino e teste.

Os dados de treino são, como o próprio nome diz, utilizados para treinar (ajustar) o algoritmo, enquanto os dados de testes são utilizados para verificar a acurácia dele, comparando o valor calculado para os testes com os valores reais.

DICA

Para separar as amostras em treino e teste, o *Scikit-Learn* disponibiliza uma função chamada *train_test_split*, que recebe como primeiro parâmetro uma lista com as *características* e segundo parâmetro uma lista com os *rótulos*.

Essa função retorna quatro novas listas:

De treino;

De teste das características;

De treino;

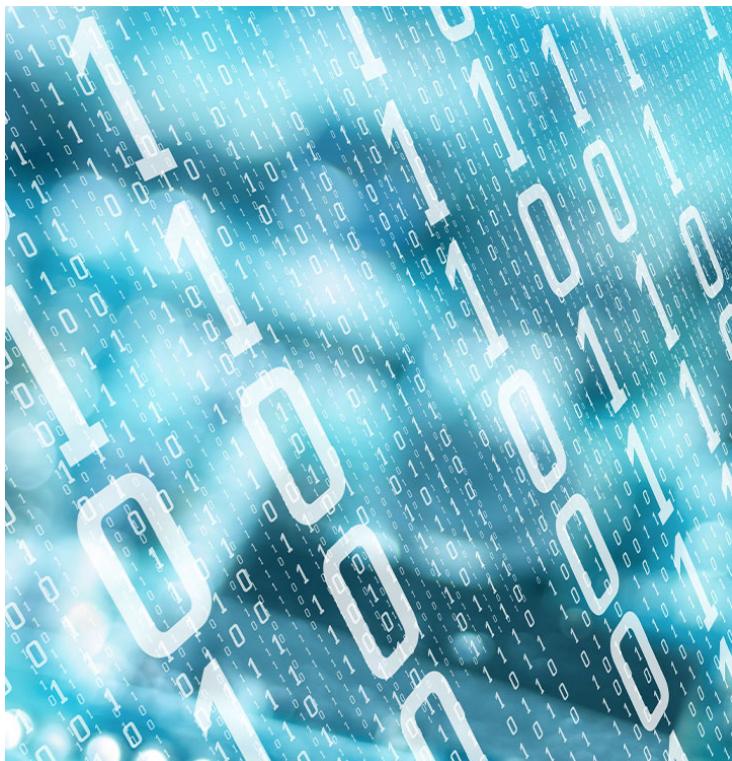
De teste dos rótulos.

Observe a linha 19, onde utilizamos essa função para gerar quatro novas variáveis: características para treino (*carac_treino*); características para teste (*carac_teste*); rótulos para treino (*rot_treino*); e rótulos para teste (*rot_teste*).

Com a etapa de pré-processamento concluída, o próximo passo é treinar um algoritmo de classificação com os dados de treino. Para isso, criamos uma instância do classificador *DecisionTree* passando como parâmetro a profundidade máxima da árvore, ou seja, o número máximo de níveis, ou camadas, a partir do nó raiz, que a árvore encontrada poderá ter (linha 24).

VEREMOS COMO A ÁRVORE FICOU AO TÉRMINO DO SCRIPT. ESSE OBJETO SERÁ UTILIZADO PARA TREINAR (AJUSTAR) A ÁRVORE DE DECISÃO. PARA REALIZAR O TREINAMENTO (FIT), PRECISAMOS PASSAR OS PARÂMETROS X E Y, QUE CONTERÃO AS CARACTERÍSTICAS DE TREINO E RÓTULOS DE TREINO RESPECTIVAMENTE (LINHA 25).

Após treinado o algoritmo, vamos utilizar o método *predict* do objeto árvore, passando como argumento as características para teste. Como resultado, receberemos uma lista com os rótulos preditos (linha 27).



Fonte: PabloLagarto/Shutterstock

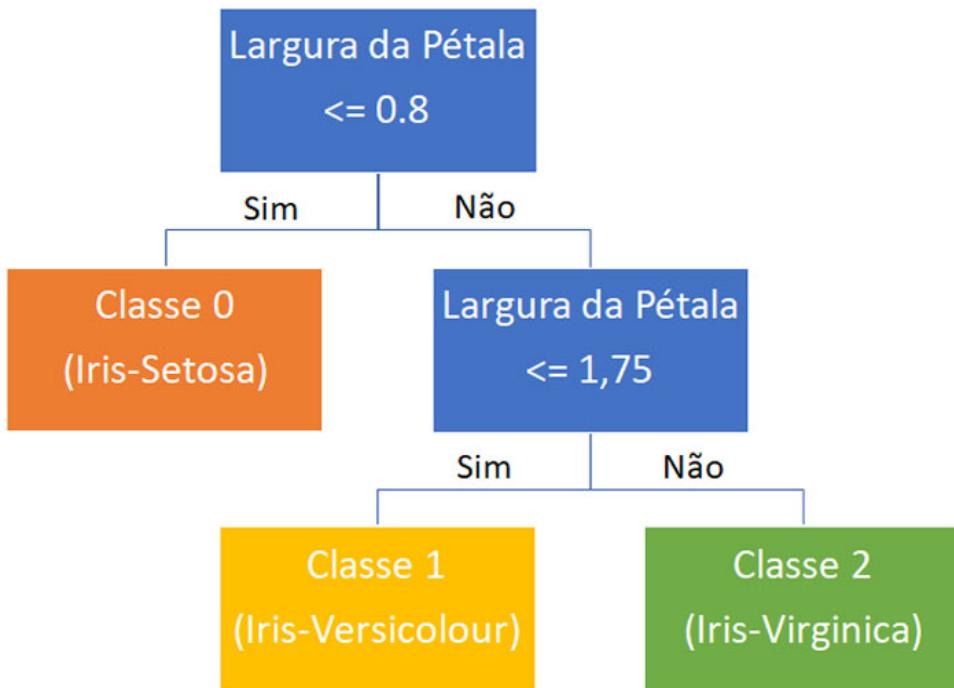
Esse resultado será utilizado como parâmetro para a função *accuracy_score*, que calcula a acurácia do classificador, comparando os resultados preditos com os resultados reais (linha 28).

ANALOGAMENTE, FAREMOS O TREINAMENTO DE UM ALGORITMO DE CLASSIFICAÇÃO UTILIZANDO O SVM, POR MEIO DA CLASSE SVC (*SUPPORT VECTOR CLASSIFICATION*) EM QUE UTILIZAREMOS OS VALORES PADRÃO DO CONSTRUTOR PARA O CLASSIFICADOR (LINHA 30 A 34).

No pós-processamento, vamos imprimir a acurácia de cada classificador (linha 37 e 38) e uma representação textual da árvore, utilizando a função `export_text` (linha 41).

Observe que a acurácia do classificador SVC foi ligeiramente melhor que da árvore de decisão, 0,97 contra 0,92 da árvore.

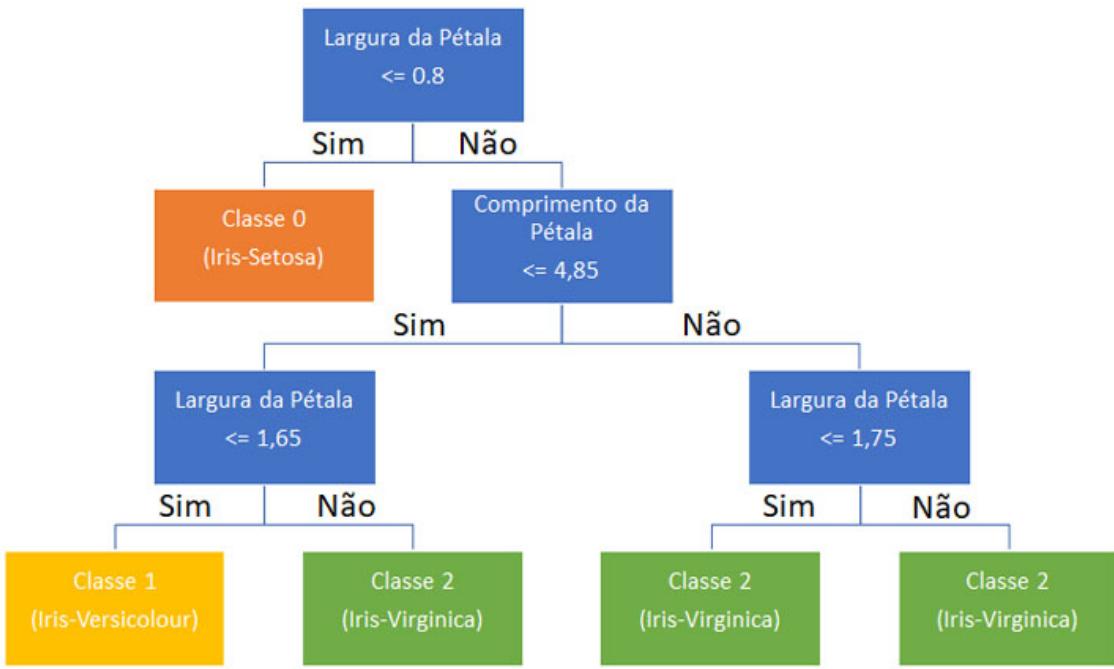
Uma representação gráfica da árvore de decisão gerada pode ser vista na **figura 12**.



Fonte: Autor

📷 **Figura 12** - Representação da árvore de decisão com profundidade 2

Alterando a profundidade da árvore para 3 e executando novamente o programa, encontramos uma acurácia de 0,97 e a seguinte árvore é exibida na **figura 13**:



Fonte: Autor

Figura 13 - Representação da árvore de decisão com profundidade 3

Durante o treinamento de algoritmos, devemos experimentar diferentes parâmetros, a fim de encontrar o melhor resultado.

NÃO SUPERVISIONADO – AGRUPAMENTO

O OBJETIVO DE UM ALGORITMO DE AGRUPAMENTO É REUNIR OBJETOS DE UMA COLEÇÃO QUE MANTENHAM ALGUM GRAU DE AFINIDADE. É UTILIZADA UMA FUNÇÃO PARA MAXIMIZAR A SIMILARIDADE DE OBJETOS DO MESMO GRUPO E MINIMIZAR ENTRE ELEMENTOS DE OUTROS GRUPOS.

Exemplos de algoritmo de agrupamento são *k-means* (*k-medias*) e *mean-shift*.

No próximo exemplo (**Código 25**), vamos utilizar o algoritmo *k-medias* para gerar grupos a partir do *dataset* de flor de íris. Porém, como o agrupamento é um algoritmo **não supervisionado** não utilizaremos os rótulos para treiná-lo. O algoritmo vai automaticamente separar as amostras em grupos, que serão visualizados em um gráfico.

Na etapa de pré-processamento, vamos começar pela **coleta e integração** que é a obtenção do *dataset* de flores utilizando a função *load_iris()* (linha 8).

Na linha 10, separamos as características das flores na variável *características*. Lembrando que as características das flores são: comprimento da sépala (índice 0), largura da sépala (índice 1), comprimento da pétala (índice 2) e largura da pétala (índice 3).

NA ETAPA DE MINERAÇÃO DE DADOS, VAMOS TREINAR O ALGORITMO DE AGRUPAMENTO *K-MEDIAS* COM AS CARACTERÍSTICAS DAS FLORES. PARA ISSO, CRIAMOS UMA INSTÂNCIA DA CLASSE *KMEANS* PASSANDO COMO PARÂMETRO O NÚMERO DE GRUPOS (OU CLASSES) QUE DESEJAMOS QUE O ALGORITMO IDENTIFIQUE (*N_CLUSTERS*) (LINHA 13).

Passamos o número 3, pois sabemos que são 3 classes de flor, mas poderíamos alterar esse valor. O objeto *grupos* criado será utilizado para treinar (ajustar) o algoritmo. Para realizar o treinamento (*fit*), precisamos passar apenas parâmetros X, que conterá as características das flores (linha 14).

Após o treino, podemos utilizar o atributo *labels_* do objeto *grupos* para retornar uma lista com o índice do grupo ao qual cada amostra pertence. Como o número de grupos (*n_clusters*) é 3, o índice varia entre: 0, 1 e 2.

Código 25 - Script agrupamento.py

```

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.cluster import KMeans
from sklearn.datasets import load_iris

##### Pré-processamento #####
# Coleta e Integração
iris = load_iris()

características = iris.data

##### Mineração #####
grupos = KMeans(n_clusters=3)
grupos.fit(X=características)
labels = grupos.labels_ # indice do grupo ao qual cada amostra pertence

##### Pós-processamento #####
fig = plt.figure(1)
ax = Axes3D(fig)
ax.set_xlabel('Comprimento Sépala')
ax.set_ylabel('Largura Sépala')
ax.set_zlabel('Comprimento Pétala')
ax.scatter(características[:, 0], características[:, 1], características[:, 2], c=grupos.labels_, edgecolor='k')

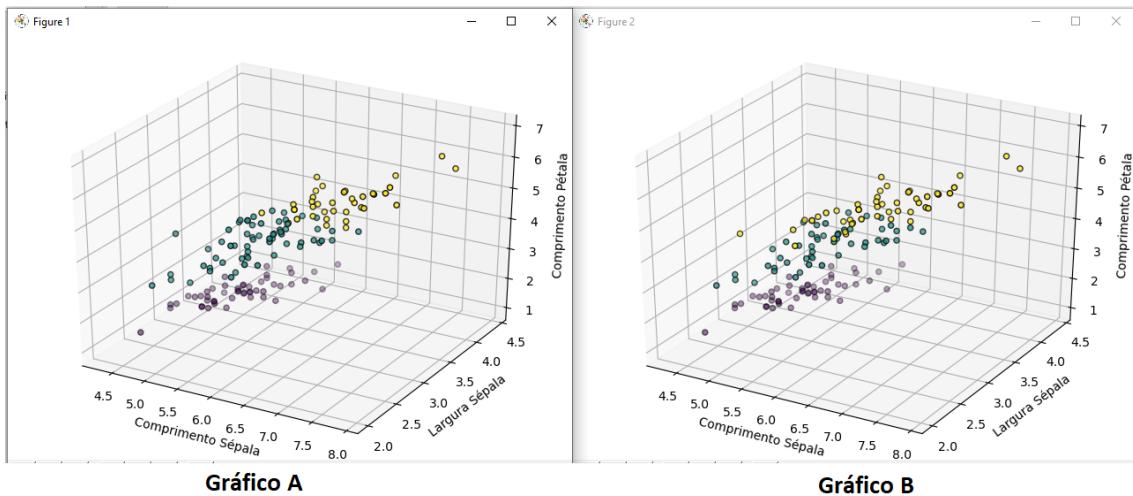
target = iris.target
fig = plt.figure(2)
ax = Axes3D(fig)
ax.set_xlabel('Comprimento Sépala')
ax.set_ylabel('Largura Sépala')
ax.set_zlabel('Comprimento Pétala')
ax.scatter(características[:, 0], características[:, 1], características[:, 2], c=target, edgecolor='k')

plt.show()

```

A partir da linha 12, temos a etapa de pós-processamento, em que utilizamos a biblioteca *matplotlib* para exibir dois gráficos, onde objetos do mesmo grupo apresentam a mesma cor.

O gráfico da **figura 14 A** contém os resultados do agrupamento e o gráfico **figura 14 B** contém os resultados reais da amostra.



Fonte: Autor

▣ **Figura 14 A - Resultado do agrupamento e Figura 14 B - Resultados reais da amostra**

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. DE ACORDO COM O PROCESSO DE DESCOBERTA DE CONHECIMENTO EM BASE DE DADOS (KDD) E ANALISANDO AS ASSERTIVAS A SEGUIR, QUAIS ATIVIDADES PODEM FAZER PARTE DA ETAPA DE PRÉ-PROCESSAMENTO?

COLETA E INTEGRAÇÃO.

CODIFICAÇÃO.

CONSTRUÇÃO DE ATRIBUTOS.

VISUALIZAÇÃO DOS DADOS.

AGORA, ASSINALE A ALTERNATIVA CORRETA:

- A) I e II**
- B) I, II e III**
- C) I, III e IV**
- D) II, III e IV**

2. EM ALGUMAS SITUAÇÕES, PRECISAMOS TRANSFORMAR UM ATRIBUTO OU CARACTERÍSTICA DE UMA AMOSTRA DE CATEGORIA PARA UM NÚMERO. QUAL O NOME DESSA ATIVIDADE?

- A) Coleta e integração.**
- B) Codificação.**
- C) Construção de atributos.**
- D) Partição dos dados.**

GABARITO

1. De acordo com o processo de descoberta de conhecimento em base de dados (KDD) e analisando as assertivas a seguir, quais atividades podem fazer parte da etapa de pré-processamento?

Coleta e Integração.

Codificação.

Construção de atributos.

Visualização dos dados.

Agora, assinale a alternativa correta:

A alternativa "B" está correta.

Todas fazem parte do pré-processamento, exceto a **visualização**, que faz parte do pós-processamento.

2. Em algumas situações, precisamos transformar um atributo ou característica de uma amostra de categoria para um número. Qual o nome dessa atividade?

A alternativa "B" está correta.

A codificação categórico-numérica transforma *string* em números.

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Como visto neste tema, a linguagem funcional pode resolver alguns problemas relacionados à execução do programa, chamados de efeitos colaterais.

Mesmo que não utilizemos todas as diretrizes da programação funcional, como imutabilidade dos dados, é uma boa prática utilizarmos funções puras, de forma a evitar a dependência do estado da aplicação e alteração de variáveis fora do escopo da função.

Em Python, podemos criar tanto programas concorrentes quanto paralelos. Apesar do GIL (*Global Interpreter Lock*) permitir executar apenas uma *thread* de cada vez por processo, podemos criar múltiplos processos em uma mesma aplicação.

Com isso, conseguimos tanto gerar códigos concorrentes (múltiplas *threads*), quanto paralelos (múltiplos processos).

Apesar do Python não ser a primeira escolha para desenvolvimento *web*, mostramos como é simples a utilização do *framework* Flask para essa finalidade. Assim como o Flask, outros *frameworks* mais completos, como Django, facilitam o desenvolvimento desse tipo de aplicação. Sites como Instagram e Pinterest utilizam a combinação Python+Django, mostrando que essa combinação é altamente escalável!

A utilização do Python como ferramenta para ciência de dados evolui a cada dia. As principais bibliotecas para análise e extração de conhecimento de base de dados, mineração de dados e aprendizado de máquinas têm uma implementação em Python.

Apesar de não ter sido mostrado, neste tema, o console do Python é muito utilizado para realizar experimentações nesta área.

Para ouvir um podcast sobre o assunto, acesse a versão online deste conteúdo.



REFERÊNCIAS

DICIONÁRIO ONLINE DE PORTUGUÊS. **Definições e significados de mais de 400 mil palavras.** Todas as palavras de A a Z. 2009-2020.

FAYYAD, U. M.; PIATETSKY-SHAPIRO, G.; SMYTH, P.; UTHURUSAMY, R. (Eds.). **From data mining to knowledge discovery in databases.** AI Magazine, v. 17, n. 3, p. 37-54, 1996.

FLASK. **Web development**, one drop at a time. 2010.

HAND, D.; MANNILA, H.; SMYTH, P. **Principles of data mining**. Massachusetts: MIT Press, 1991.

HUNT, J. **A beginners guide to Python 3 programming**. Basileia: Springer, 2019.

MATPLOBIT. **Visualization with Python**. John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team. 2012-2020.

PANDAS. **Sponsored project of NumFOCUS**. 2015.

PYTHON. **Python Software Foundation**. Copyright 2001-2020.

SCIKIT-LEARN. **Machine Learning in Python**. 2007.

W3TECHS. **Web Technology Surveys**. 2009-2020.

EXPLORE+

Acesse a documentação oficial do Python sobre as funções e os operadores que se integram bem ao paradigma de programação funcional, para aprender mais sobre o assunto.

Pesquise no *site* da Jinja2 e conheça mais detalhadamente as funcionalidades da linguagem de modelos (*templates*) para Python.

CONTEUDISTA

Frederico Tosta de Oliveira

 CURRÍCULO LATTES