



Python em outros paradigmas

Prof. Frederico Tosta de Oliveira

Prof. Kleber de Aguiar

Descrição

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

Linguagem funcional no Python

Identificar a linguagem funcional e sua utilização em Python.

Módulo 2

Computação concorrente em Python

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

Módulo 3

Desenvolvimento Web com Python

Identificar o Python como ferramenta para desenvolvimento Web.

Módulo 4

Ciência de Dados em Python

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.

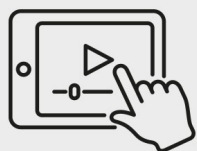
Aprenderemos também conceitos relacionados à computação concorrente e à computação paralela, suas diferenças e como podem ser implementadas em Python.

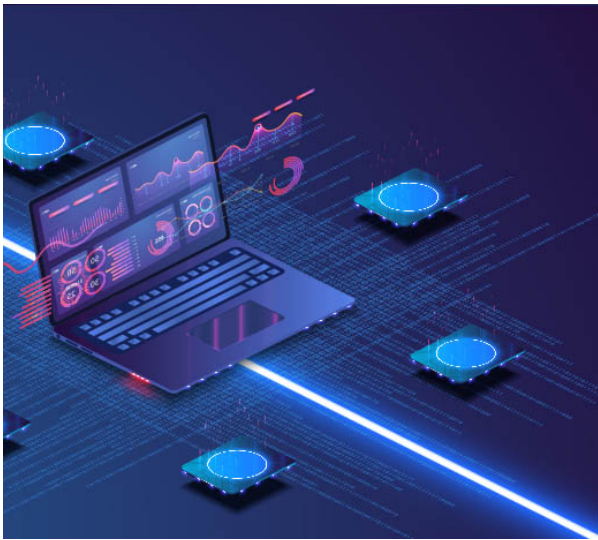
Embora o Python não seja a primeira escolha como linguagem de programação para servidores Web, é possível encontrar diversos frameworks que facilitam, e muito, a sua utilização para tal finalidade. Alguns são minimalistas, como o — Flask — e permitem criar servidores Web escritos em Python em minutos.

Em contrapartida, essa linguagem de programação é a escolha número 1 para a ciência de dados. Neste tema, conheceremos 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.

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





1 - Linguagem funcional no Python

Ao final deste módulo, você será capaz de identificar a linguagem funcional e sua utilização em Python.

Visão geral

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 as seguintes regras:

ISP

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.

As funções 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.

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 imprevisível 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.

Relação entre funções puras e dados imutáveis

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**, nos dois emuladores a seguir:

Exercício 1

 TUTORIAL  COPIAR

Python3

```
1 # script funcao1.py
2 valor = 20
3
4 def multiplica(fator):
5     global valor
6     valor = valor * fator
```

null

null



Exercício 1

TUTORIAL COPIAR

Python3

```
1 # script funcao2.py
2 valor = 20
3
4 def multiplica(valor, fator):
5     valor = valor * fator
6     print(valor)
```

null

null



Agora observe a seguir o que ocorreu em cada um dos scripts:

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 6), que é impresso em seguida (linha 7).

Por exemplo, se chamarmos a função `multiplica` pela primeira vez, passando o valor **3** como parâmetro, obteremos a saída "Resultado 60". Como modificamos a própria variável global `valor` no corpo da função, ao chamarmos novamente a função `multiplica` passando novamente o **3** como parâmetro, obtemos um resultado de saída diferente: "Resultado 180".

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

Verifique por si mesmo! No primeiro emulador, delete a instrução `pass` (linha 10) e dentro da função `main` chame duas vezes a função `multiplica` com um mesmo valor como parâmetro (exemplo: `multiplica(3)`). Clique em Executar.

funcao2.py

Utilizamos a variável `valor` e a função `numero` como parâmetros para a função `multiplica`. O valor da variável `numero` será recebido via teclado, por meio da função nativa `input`.

As duas vezes em que executamos essa função (linhas 10 e 11), retornarão o mesmo valor. Por exemplo, se o valor de `numero` for **3**, ambas as saídas do script serão: "Resultado 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 **funcao3.py** e **funcao4.py** a seguir, em que passamos a lista valores como argumento para a função altera_lista. Lembre-se de 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. Nos emuladores seguintes, informe no campo Input os números inteiros a serem atribuídos à lista valores e clique em Executar para que os scripts sejam executados.

Os elementos devem ser separados por 1 espaço em branco, por exemplo: 10 20 30.

Exercício 1

TUTORIAL

COPIAR

Python3

1

script funcao3.py

2

captando os valores do campo Input

3

valores = input()

4

separando os valores pelo espaço em branco e

5

transformando-os em uma lista de inteiros

6

valores = valores.split(' ')
valores = [int(valor) for valor in valores]

null

null



Exercício 1

TUTORIAL

COPIAR

Python3

1

script funcao4.py

2

3

captando os valores do campo Input

4

5

valores = input()

6

null

null



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 9), alteramos também a variável global valores.

Com isso, ao chamar a mesma função duas vezes (linhas 13 e 14), com, teoricamente, o mesmo parâmetro, obtemos um efeito indesejável, resultando em saídas diferentes, como pode ser observado no campo Console do respectivo emulador.

No exemplo do script **funcao4.py**, muito similar ao script **funcao3.py**, ao invés de alterarmos o valor do próprio parâmetro, criamos uma cópia dele (linha 5), sendo assim, não alteramos a variável valores e obtemos o mesmo resultado para as duas chamadas da função (linhas 10 e 11), fato observado no campo Console do emulador que contém o script **funcao4.py**.

Efeitos indesejáveis

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.

feito colateral

É quando a função faz alguma operação que não seja computar a saída a partir de uma entrada. 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.

Atenção!

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.

Outros tipos de função

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 a seguir, script **funcao5.py**, 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**). Veja o seguinte código:

Exercício 1



Python3

```
1 # script funcao5.py
2 def multiplicar_por(multiplicador):
3     def multi(multiplicando):
4         return multiplicando * multiplicador
5     return multi
6
```

null

null



Dentro da função “pai” `multiplicar_por`, definimos a função `multi` (linha 3), 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 8), definimos a função interna `multi` como:

PYTHON



```
1 def multi(multiplicando):
2     return multiplicando * 10
```

Essa função é retornada e armazenada na variável `multiplicar_por_10` (linha 8), 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 9 e 10), 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 12), que recebe uma referência para a função:

PYTHON



```
1 def multi(multiplicando):
2     return multiplicando * 5
```


Com isso, podemos utilizar a função `multiplicar_por_5` para multiplicar um número por 5 (linhas 13 e 14).

Clique no emulador anterior e observe a saída do programa em seu campo console.

Funções lambda

Assim como em outras linguagens, o Python permite a criação de funções anônimas. Estas 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`.

Atenção!

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` (script `funcao5.py`), podemos trocar a função `multi` por uma função lambda.



Atividade discursiva

Que tal tentar implementar essa alteração na função `multiplicar_por`? No emulador a seguir, insira o seu código no lugar da palavra reservada `pass` (linha 3):

Exercício 1

 TUTORIAL  COPIAR

Python3

```
1 # script anonima.py (versão alterada do script funcao5.py)  
2 def multiplicar_por(multiplicador):  
3     pass  
4  
5 def main():  
6     ...
```

null

null



Digite sua resposta aqui...

Exibir solução ▾

Após tentar implementar a sua alteração, você pode conferir o script alterado a seguir:

Python



```
1 # script anonima.py (versão alterada do script funcao5.py)
2 def multiplicar_por(multiplicador):
3     return lambda multiplicando: multiplicando * multiplicador
4
5 def main():
6     multiplicar_por_10 = multiplicar_por(10)
7     print(multiplicar_por_10(1))
8     print(multiplicar_por_10(2))
9
10    multiplicar_por_5 = multiplicar_por(5)
11    print(multiplicar_por_5(1))
12    print(multiplicar_por_5(2))
13
14 if __name__ == "__main__":
```

Ao executar o script anonima.py, deve-se obter o mesmo resultado mostrado anteriormente: 10, 20, 5 e 10.

Boa prática de programação

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.

Maps

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.

Nos exemplos a seguir, vamos criar três scripts **funcao_iterable.py**, **funcao_map.py** e **funcao_map_lambda.py**. Todos executam a mesma operação. Recebem uma lista e triplicam cada item, gerando uma **nova** lista com os valores triplicados:

Exercício 1

TUTORIAL COPIAR

Python3

```
1 # script funcao_iterable.py
2 lista = [1, 2, 3, 4, 5]
3
4 def triplica_itens(iterable):
5     lista_aux = []
6     for item in iterable:
```

null

null



Exercício 1

TUTORIAL COPIAR

Python3

```
1 # script funcao_map.py
2 lista = [1, 2, 3, 4, 5]
3
4 def triplica(item):
5     return item * 3
6
```

null

null



Exercício 1

TUTORIAL COPIAR

Python3

```
1 # script funcao_map_lambda.py
2 lista = [1, 2, 3, 4, 5]
3
4 nova_lista = map(lambda item: item * 3, lista)# função lambda
5
6
```

null

null



Veja a seguir o que ocorreu em cada um dos scripts:

funcao_iterable.py



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

Essa função é chamada com o argumento lista (linha 11) e o resultado é impresso (linha 12).

funcao_map.py



Definimos a função triplica, que triplica e retorna o item passado como parâmetro (linhas 4 e 5). É utilizada, assim como a variável lista, como argumentos para a função map (linha 8).

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 9).

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 9).

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 4). Esta vai aplicar a função lambda em cada item da lista, retornando um novo **iterável** que foi impresso posteriormente (linha 7).

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.

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 (verdadeiro ou falso) para cada elemento do iterável.

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.

Nos exemplos a seguir, funções **função_filtro_iterable.py**, **função_filter.py** e **função_filter_lambda.py**, 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.

Exercício 1

 TUTORIAL  COPIAR

Python3



```
1 # script funcao_filtro_iterable.py
2 lista = [1, 2, 3, 4, 5]
3
4 def impares(iterable):
5     lista_aux = []
6     for item in iterable:
```

null

null



Exercício 1

 TUTORIAL  COPIAR

Python3

```
1 # script funcao_filter.py
2 lista = [1, 2, 3, 4, 5]
3
4 def impar(item):
5     return item % 2 != 0
6
```

null

null



Exercício 1

TUTORIAL

COPIAR

Python3

```
1 # script funcao_filter_lambda.py
2 lista = [1, 2, 3, 4, 5]
3
4 nova_lista = filter(lambda item: item % 2 != 0, lista)
5
6
```

null

null



Veja o que ocorreu com cada um dos scripts:

função_filtro_iterable.py

Definimos uma função chamada `ímpares`, que recebe um iterável como parâmetro (linha 4), cria uma lista auxiliar para garantir imutabilidade (linha 5), percorre os itens do iterável passados como parâmetros (linha 6), adiciona os itens ímpares à lista auxiliar (linhas 7 e 8) e retorna a lista auxiliar (linha 9).

Essa função é chamada com o argumento `lista` (linha 12) e o resultado é impresso (linha 13).

função_filter.py

Definimos a função `ímpar`, que retorna `true` se o item passado como parâmetro é ímpar ou `false` caso contrário (linhas 4 e 5). Utilizamos essa função, assim como a variável `lista`, como argumentos para a função `filter` (linha 8).

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 9).

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 9).

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 4). Esta vai aplicar a função lambda em cada item da lista, retornando um novo **iterável** que foi impresso posteriormente (linha 7).

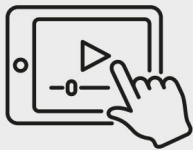
Todos os scripts geraram o mesmo resultado: [1, 3, 5].



Aplicando programação funcional com Python

Neste vídeo, abordaremos conceitos que foram aprendidos na computação funcional.

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



Exercícios de programação funcional em Python

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



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 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

E

II e IV

Parabéns! A alternativa D está correta.

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

Questão 2

Qual é o resultado impresso pelo programa a seguir?

Scriptlambda1.Py

```
1 minha_funcao = lambda x: xx ** 2
2 resultado = minha_funcao(4)
3 print(resultado)
```

A

4

B

8

C

16

D

32

E

64

Parabéns! A alternativa C está correta.

A função criada retorna o valor do argumento ao quadrado.



2 - Computação concorrente em Python

Ao final deste módulo, você será capaz de definir os conceitos de computação concorrente e sua utilização em Python.

Visão geral

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.



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.

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.

concorrente

O dicionário on-line, Dicio, define a palavra “concorrente” como: “Simultâneo; o que acontece ao mesmo tempo que outra coisa [...]”.

Conceitos de programa e processo

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.

Conceitos de 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

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.

Conceito de Threads

Threads e processos

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.

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

Curiosidade

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.

Antes de falarmos sobre cada abordagem em Python, vamos falar sobre o global interpreter lock (GIL). No [CPython](#), GIL existe para proteger o acesso aos objetos da linguagem.

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.

Python

É a implementação principal da linguagem de programação Python, escrita em linguagem C.

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 concorrente. Sem a “trava” do GIL, uma operação simples de atribuição de variável poderia gerar um dado inconsistente caso duas threads atribuissem 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.

Quando temos, porém, 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

No script principal.py a seguir, vamos criar uma thread e um processo que executam a mesma função. Na linha 9, 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`.

Atenção!

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.

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 10, enviamos o comando para a thread iniciar sua execução, chamando o método `start()` do objeto `t` do tipo `thread`, como pode ser observado a seguir:

Python3

```
1 # script principal.py
2 from threading import Thread
3 from multiprocessing import Process
4
5 def funcao_a(nome):
6     ...
```

null

null



A criação do processo é praticamente igual, porém utilizando a classe process, conforme a linha 12. Para iniciar o processo, chamamos o método start() na linha 13.

Verifique a saída desse script no console do emulador anterior, clicando no botão Executar.

Múltiplas threads e processos

No exemplo a seguir, `scripts threads_var.py` e `processos_var.py`, vamos criar múltiplas 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.

Neste exemplo, a função que será paralelizada é a funcao_a (linha 8). 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 6.

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).

Atenção!

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

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. Veja os códigos a seguir:

Script Threads_var.Py



```
1 # script threads_var.py
2 from threading import Thread, Lock
3 from multiprocessing import Process
4 import time
```

```

5  minha_lista = []
6
7  def funcao_a(indice):
8      for i in range(100000):
9          minha_lista.append(1)
10         print("Termino thread ", indice)
11
12 def main():
13     tarefas = []
14

```

Script Processos_var.Py

```

1  # script processos_var.py
2  from threading import Thread, Lock
3  from multiprocessing import Process
4  import time
5
6  minha_lista = []
7
8  def funcao_a(indice):
9      for i in range(100000):
10         minha_lista.append(1)
11         print("Termino thread ", indice)
12
13 def main():
14     tarefas = []

```

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).

Nos seguintes códigos, temos o resultado de cada script, que são:

Saída Do Script Threads_var.Py

```

1  C:\Users\ftoli\PycharmProjects\estacio_ead\venv\Scripts\python.e
2  Termino thread Termino thread 0 1
3
4  Termino thread Termino thread 23Termino thread 4
5
6  Termino thread 5
7  Termino thread 6
8  Termino thread 7
9  9Termino thread Antes de guardar o termino! Termino thread 9
10 970012
11 8
12 Apos guardar o termino! 1000000
13
14 Process finished with exit code 0

```

Saída Do Script Processos_var.Py

```
1 C:\Users\ftoli\PycharmProjects\estacio_ead\venv\  
2 Antes guardar o termino! 0  
3 Termino thread 0  
4 Termino thread 2  
5 Termino thread 1  
6 Termino thread 4  
7 Termino thread 3  
8 Termino thread 5  
9 Termino thread 7  
10 Termino thread 6  
11 Termino thread 8  
12 Termino thread 9  
13 Apos guardar o termino! 0  
14
```

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` incrementará a variável global `contador`. Observe o código a seguir, script `threads_inc.py` e verifique o resultado impresso no console seguinte:

Script Threads_inc.Py

```
1 # script threads_inc.py  
2 from threading import Thread, Lock  
3 from multiprocessing import Process  
4 import time  
5  
6 contador = 0  
7  
8 def funcao_a(indice):  
9     global contador  
10    for i in range(1000000):  
11        contador += 1  
12    print("Termino thread ", indice)  
13  
14 def main():
```

Saída Do Script Threads_inc.Py

```
1 C:\Users\ftoli\PycharmProjects\estacio_ead\venv\S  
2 Termino thread 1  
3 Termino thread Termino thread 3 2  
4
```

```

5 Termino thread 0 Termino thread
6 Antes de aguardar o termino! Termino thread 4
7 Termino thread 6
8 82235554
9
10 Termino thread Termino thread 9
11 Termino thread 7
12
13 Apos aguardar o termino! 3316688
14

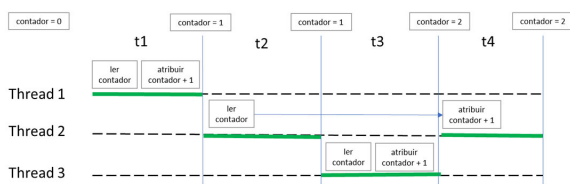
```

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.

Por que não aconteceu isso no 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 11), 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 imagem a seguir, 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:



Threads em execução.

Cada thread é executada em um determinado tempo t . Em t_1 , a thread1 lê e incrementa o contador, que passa a valer 1. Em t_2 , a thread2 lê o contador (valor 1). Em t_3 , a thread3 lê e incrementa o contador, que passa a valer 2. Em t_4 , 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 script `threads_inc_lock.py` a seguir:

Script `Threads_inc_lock.py`

```

1 # script threads_inc_lock.py
2 from threading import Thread, Lock
3 from multiprocessing import Process
4 import time
5
6 contador = 0

```

```

7  lock = Lock()
8  print_lock = Lock()
9
10 def funcao_a(indice):
11     global contador
12     for i in range(1000000):
13         lock.acquire()

```

Primeiramente, definimos a variável global lock utilizando o construtor lock(), também do módulo threading (linha 7). 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 13), de forma a garantir exclusividade na operação de incremento (linha 14).

Logo após o incremento, liberamos a trava utilizando o método release (linha 15). Com apenas essa modificação, garantimos que o resultado estará correto. Podemos verificar o resultado no console abaixo.

Saída Do Script Threads_inc_lock.Py

```

1  C:\Users\ftoli\PycharmProjects\estacio
2  Antes de aguardar o termino! 73844
3  Termino thread 1
4  Termino thread 7
5  Termino thread 8
6  Termino thread 9
7  Termino thread 2
8  Termino thread 3
9  Termino thread 0
10 Termino thread 4
11 Termino thread 6
12 Termino thread 5
13 Após aguardar o termino! 10000000
14

```

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 17 com outra trava, print_lock, criada na linha 8.

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 a seguir, **script processos.py**, criaremos uma variável do tipo inteiro e a compartilharemos entre os processos. Essa variável fará o papel de um contador e a função paralelizada vai incrementá-la:

Script Processos.Py

```

1  # script processos.py
2  from threading import Thread
3  from multiprocessing import Process, Value
4
5  def funcao_a(indice, cont):
6      for i in range(100000):
7          with cont.get_lock():

```



```
8         cont.value += 1
9         print("Termino processo ", indice)
10
11 def main():
12     contador = Value('i', 0)
13     tarefas = []
14
```

Saída Do Script Processos.Py

```
1 C:\Users\ftoli\PycharmProjects\estacio_e
2 Antes de aguardar o termino! 0
3 Termino processo 2
4 Termino processo 1
5 Termino processo 0
6 Termino processo 3
7 Termino processo 4
8 Termino processo 5
9 Termino processo Termino processo 7
10 6
11 Termino processo 8
12 Termino processo 9
13 Após aguardar o termino! 1000000
14
```

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 12).

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 índice da funcao_a recebe o valor da variável índice e o parâmetro cont recebe uma referência para a variável contador (linha 15).

Isso já é o suficiente para termos acesso à 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.

Curiosidade

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 7).

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.



Utilizando Python na programação concorrente

Neste vídeo, abordaremos programação paralela usando o conceito de Thread em Python

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



Exercícios de Python na programação concorrente

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



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 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

E

I, II e III

Parabéns! A alternativa C está correta.

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

Questão 2

Qual o valor impresso pelo script a seguir:?

Script Processos.Py

```
1 # script processos.py
2 from threading import Thread
3
4 minha_lista = []
5
6 def adiciona():
7     for i in range(100):
8         minha_lista.append(1)
9
10 def main():
11     tarefas = []
12
13     for indice in range(10):
14         t = Thread(target=adiciona)
```

Script processos.py.

A

100

B

1000

C

2000

D

10000

E

20000

Parabéns! A alternativa B está correta.

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



3 - Desenvolvimento Web com Python

Ao final deste módulo, você será capaz de identificar o Python como ferramenta para desenvolvimento Web.

Visão geral

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 têm muita representatividade na Web hoje.

Curiosidade

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. Veja cada um deles a seguir:

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 ao 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

Oferecem os recursos básicos para uma aplicação Web, como resposta à 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.

Conceito

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](#).

lask

É um micro framework Web em Python.

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`.

Começaremos 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 a seguir:

Script Flask1.Py

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route('/')
6 def ola_mundo():
7     return "Olá, mundo."
8
9 if __name__ == '__main__':
10     app.run()
```

Saída Do Script Flask1.Py-5



```
1 C:\Users\ftoli\PycharmProjects\estacio_ead\venv\Scripts\python.exe C:/Users/ftoli/Py
2 * Serving Flask app "principal" (lazy loading)
3 * Environment: production
4 WARNING: This is a development server. Do not use it in a production deployment.
5 Use a production WSGI server instead.
6 * Debug mode: off
7 * 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.

Curiosidade

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.

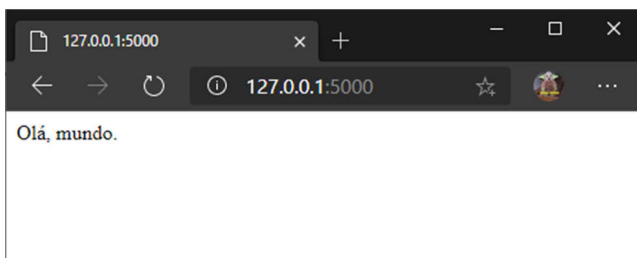
Por isso, utilize nomes com significado e de fácil memorização. No caso do Flask, as rotas são direcionadas para funções, que devem retornar algum conteúdo.

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".

Ao digitar no navegador a URL `http://127.0.0.1:5000`, será exibida a frase "Olá, mundo.", como na imagem seguinte:



Resultado do acesso a `http://127.0.0.1:5000`.

Rotas e parâmetros

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`.

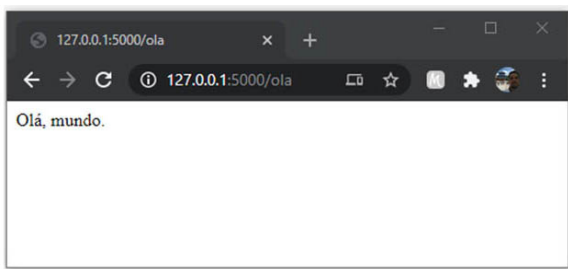
Observe o script **flask2.py** e compare com o anterior. Veja que o parâmetro do decorador `@app.route` da linha 5 agora é `/ola`:

```
Script Flask2.Py
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route('/ola')
6 def ola_mundo():
7     return "Olá, mundo."
8
9 if __name__ == '__main__':
10     app.run()
```

Se acessarmos as seguintes URLs no navegador, veremos o seguinte:

<http://127.0.0.1:5000/ola>

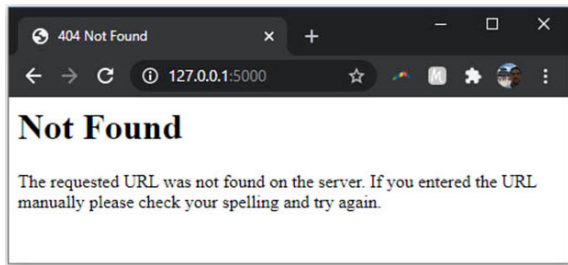
Recebemos como resposta “Olá, mundo.”, como mostrado na imagem a seguir:



Resultado do acesso de <http://127.0.0.1:5000/ola>.

<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, como mostrado na imagem a seguir:



Resultado do acesso de `http://127.0.0.1:5000`.

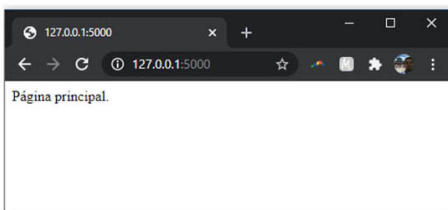
Vamos acertar nossa aplicação para criar, também, uma rota para a URL raiz da aplicação (`@app.route('/')`).

Vamos chamar a função dessa rota de `index` e criar essa rota conforme linhas 5 a 7. Veja o resultado no script **flask3.py** a seguir:

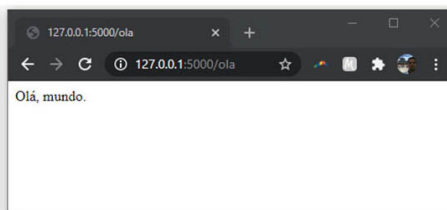
Script Flask3.Py

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route('/')
6 def index():
7     return "Página principal."
8
9 @app.route('/ola')
10 def ola_mundo():
11     return "Olá, mundo."
12
13 if __name__ == '__main__':
14     app.run()
```

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



A



B

Resultado do acesso a `http://127.0.0.1:5000` e 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 `< >` na URL da rota.

No próximo exemplo, vamos mudar a rota da função `ola_mundo` de forma que seja possível capturar e retornar o nome passado como parâmetro:

Script Flask4.Py

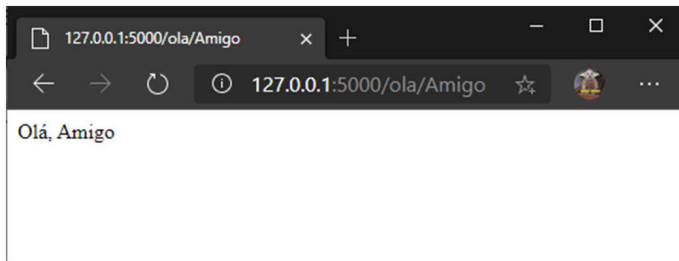
```
1 from flask import Flask
```



```
2
3 app = Flask(__name__)
4
5 @app.route('/')
6 def index():
7     return "Página principal."
8
9 @app.route('/ola/<nome>')
10     def ola_mundo(nome):
11         return "Olá, " + nome
12
13 if __name__ == '__main__':
14     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 imagem a seguir, em que acessamos a URL <http://127.0.0.1:5000/ola/Amigo>:



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 o script `flask5.py` a seguir:

Script Flask5.Py

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route('/')
6 def index():
7     return "Página principal."
8
9 @app.route('/ola/')
10 @app.route('/ola/<nome>')
11     def ola_mundo(nome="mundo"):
12         return "Olá, " + nome
13
14 if __name__ == '__main__':
```

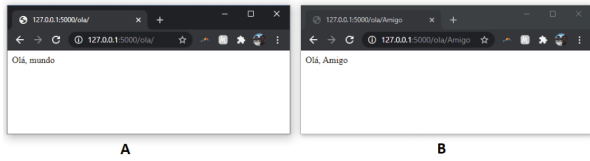
Observe que agora temos duas rotas para a mesma função (linhas 9 e 10). Em qualquer uma das URL, o usuário será direcionado para a função `ola_mundo`.

Atenção!

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 imagem A e B, respectivamente:



Resultado do acesso a `http://127.0.0.1:5000/ola/` e do acesso a `http://127.0.0.1:5000/ola/Amigo`.

Métodos HTTP e modelos

Métodos HTTP

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

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.

Observe no script `flask6.py`, 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.

Script Flask6.Py

```
1 from flask import Flask
2
3 app = Flask(__name__)
4 app.debug = True
5
6 @app.route('/')
7 def index():
8     return "Página principal."
9
10 @app.route('/ola/')
11 @app.route('/ola/<nome>')
12 def ola_mundo(nome):
13     return "Olá, " + nome
14
```

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.

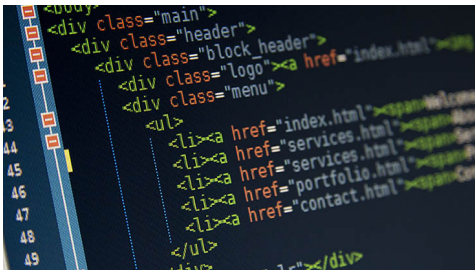
Observe que ativamos o modo debug do servidor interno do Flask (linha 4). Isso nos permite visualizar melhor os avisos e erros durante o desenvolvimento.

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 (escape) 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.

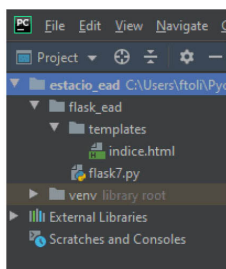


A cada requisição, podemos alterar uma mesma página de acordo com um contexto (valores de variáveis, por exemplo). Com a utilização de marcadores (tags) especiais, é possível injetar valores de variáveis no corpo do HTML, criar laços (for), condicionantes (if/else), filtros etc.

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.

Para ilustrar, no próximo exemplo (script `flask7.py`), vamos alterar nossa aplicação de forma que seja retornada uma página HTML ao se acessar a raiz da aplicação ('/').

Para isso, criaremos um arquivo chamado `indice.html` na pasta `templates`, no mesmo nível do script `flask7.py`, conforme vemos na árvore de diretório a seguir:



Árvore de diretório.

O conteúdo do modelo indice.html está exibido abaixo do script flask7.py.

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.

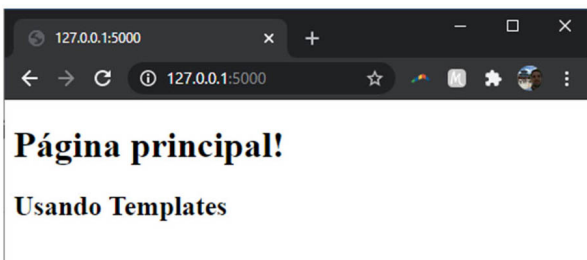
Script Flask7.Py

```
1 from flask import Flask, render_template
2
3 app = Flask(__name__)
4
5 @app.route('/')
6 def index():
7     return render_template('indice.html')
8
9 @app.route('/ola/')
10 @app.route('/ola/<nome>')
11 def ola_mundo(nome="mundo"):
12     return "Olá, " + nome
13
14 if __name__ == '__main__':
```

Modelo Indice.Html

```
1 <!DOCTYPE html>
2 <html>
3     <body>
4         <h1>Página principal!</h1>
5         <h2>Usando templates</h2>
6     </body>
7 </html>
```

Ao acessar <http://127.0.0.1:5000>, recebemos o resultado exibido na imagem a seguir:



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: {{ ... }}

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

Declarações: {% ... %}

Serve para ser 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, alteraremos a nossa aplicação de forma que o nome passado como parâmetro para a rota '/ola/' seja exibido dentro do HTML.

Para isso, vamos criar um arquivo chamado ola.html na pasta templates, conforme abaixo no script **flask8.py**. Observe que utilizamos o delimitador de expressões com a variável ({{nome_recebido}}), indicando que escreveremos o conteúdo dessa variável nesse local após a renderização (linha 4 do arquivo ola.html após o script flask8.py).

Script Flask8.Py

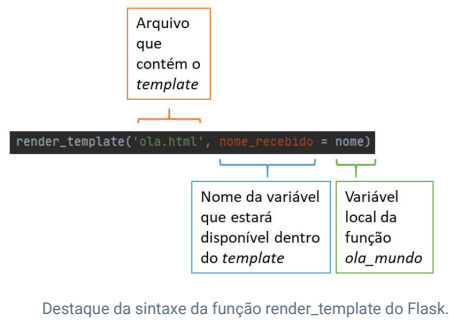
```
1 from flask import Flask, render_template
2
3 app = Flask(__name__)
4
5 @app.route('/')
6 def index():
7     return render_template('indice.html')
8
9 @app.route('/ola/')
10 @app.route('/ola/<nome>')
11 def ola_mundo(nome="mundo"):
12     return "Olá, ", nome_recebido = nome
13
14 if __name__ == '__main__':
```

Modelo Ola.Html

```
1 <!DOCTYPE html>
2 <html>
3     <body>
4         <h1>Olá, {{ nome_recebido }}</h1>
5     </body>
6 </html>
```

Além de informar a página que queremos renderizar na função `render_templates`, 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 na imagem a seguir:



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 a imagem seguinte, onde passamos a string “Amigo” para a variável `nome` e, conseqüentemente, `nome_recebido`:



Desenvolvendo aplicações Web com Python

Neste vídeo, abordaremos framework Flask para desenvolvimento de aplicações Web.

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

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

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

Exercicio1.Py

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route('/ola')
6 def ola_mundo():
7     return "Olá,mundo"
8
9 @app.route('/ola')
10 def ola_mundo(nome="mundo"):
11     return "Olá, " + nome
12
13 if __name__ == '__main__':
14     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.
Olá, EAD.
- C Olá, EAD.
- D O programa vai apresentar um erro.
- E Olá, EAD.
Olá, mundo.

Parabéns! A alternativa C está correta.

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

Questão 2

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

Exercicio2.Py

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route('/ola', methods=['POST'])
6 def ola_post():
7     return "Olá, GET"
8
9 @app.route('/ola')
10 def ola_get(nome="mundo"):
11     return "Olá, POST"
12
13 if __name__ == '__main__':
14     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.

E Olá, POST.
Olá, GET.

Parabéns! 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.



4 - Ciência de Dados em Python

Ao final deste módulo, você será capaz de identificar o Python como ferramenta para Ciência de Dados.

Visão geral

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.

Em 1996, Fayyad 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, iterativo 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

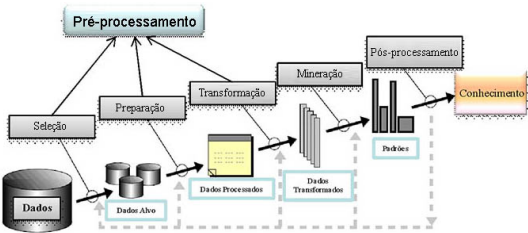


Mineração de dados



Pós-processamento

A imagem a seguir mostra todo o processo de KDD:



Visão geral dos passos que compõem 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 pré-processamento são:

- Coleta e integração

Quando é necessário que dados provenientes de diversas fontes sejam consolidados em uma única base de dados. Essa atividade é bastante encontrada na construção de data warehouses.
- Codificação

Significa transformar a natureza dos valores de um atributo. Isso 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

Pode ser necessário criar colunas em uma tabela, por exemplo, refletindo alguma transformação dos dados existentes em outras colunas, após a coleta e integração dos dados.

Limpeza de 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.
 (HAND, 2001, p.6)

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.

JA seguir, apresentaremos 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:

Não 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.

Tabela: Algoritmos durante a etapa de mineração de dados.
Frederico Tosta de Oliveira

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

Apresentaremos 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.

Algoritmos supervisionados

Supervisionado – regressão linear

Neste exemplo, utilizaremos uma série histórica fictícia de casos de dengue de uma determinada cidade e, com o auxílio do algoritmo supervisionado de regressão linear, predizeremos 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. Utilizaremos essa série histórica e aplicaremos o algoritmo de regressão linear para estimar os casos de dengue para o ano de 2018.

A seguir, temos o código do script regressao.py, o arquivo CSV (dados_dengue.csv) e a saída do console.

Script Regressão.Py

```
1 import matplotlib.pyplot as plt
2 from sklearn.linear_model import LinearRegression
3 import pandas
4
5 ##### Pré-processamento #####
6 # Coleta e Integração
7 arquivo = pandas.read_csv('dados_dengue.csv')
8
9 anos = arquivo[['ano']]
10 casos = arquivo[['casos']]
11
12 ##### Mineração #####
```

```
13 regr = LinearRegression()
```

Dados_dengue.Csv

	ano,casos
2	2017,450
3	2016,538
4	2015,269
5	2014,56
6	2013,165
7	2012,27
8	2011,156
9	2010,102
10	2009,86
11	2008,42
12	2007,79
13	2006,65
14	2005,58

Saída Do Script Regressão.Py

```
1 C:\Users\fotoli\PycharmProjects\estac
2 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, utilizaremos 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 prever 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 de casos previsto para 2018, conforme impresso no console da figura **Saída do script regressão.py** (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, como na imagem a seguir:

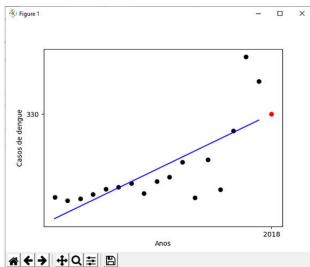


Gráfico da série dos casos de dengue.

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, utilizaremos 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 está fora do escopo deste módulo.

Confira o script a seguir, `classificação.py`, em que utilizamos esses dois algoritmos:

Script Classificacao.Py

```
1 from sklearn.datasets import load_iris, fetch_kddcup99
2 from sklearn.metrics import accuracy_score
3 from sklearn.model_selection import train_test_split
4 from sklearn.tree import DecisionTreeClassifier, export_text, plot_tree
5 from sklearn.svm import SVC
6
7 ##### Pré-processamento #####
8 # Coleta e Integração
9 iris = load_iris()
```

```
10
11 caracteristicas = iris.data
12 rotulos = iris.target
13
14 print(f"Características: {caracteristicas}\nRótulos: {rotulos}")
```

Saída Do Script Classificacao.Py

```
1 C:\Users\fotoli\PycharmProjects\estacio_ead\
2 Características:
3 [[5.1 3.5 1.4 0.2]
4  [4.9 3.  1.4 0.2]]
5 Rótulos:
6 [0 0]
7 #####
8 Acurácia Árvore de Decisão: 0.92105
9 Acurácia SVM: 0.97368
10 #####
11 Estrutura da árvore
12 |--- petal width (cm) <= 0.80
13 |   |--- class: 0
14 |--- petal width (cm) > 0.80
```

Veja a seguir cada etapa do pré-processamento:

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). Essa função retorna um objeto onde podemos acessar as características das flores pelo atributo `data` e os rótulos, ou classes das flores, pelo atributo `target`.

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 dessa lista no console. Cada um dos quatro elementos corresponde ao comprimento da sépala, largura da sépala, comprimento da pétala e largura da pétala, respectivamente.

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-Versicolor*; 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()`.

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).

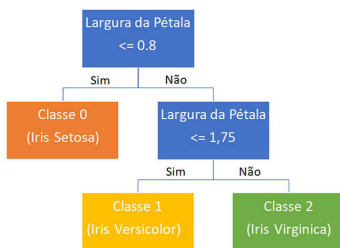
Esse resultado será utilizado como parâmetro para a função `accuracy_score`, que calcula a acurácia do classificador, comparando os resultados preditos com os resultados reais (linha 28).

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 (linhas 30 a 34).

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

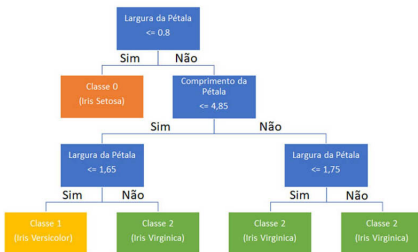
Observe que a acurácia do classificador SVC foi ligeiramente melhor que da árvore de decisão, 0,97 contra 0,92 da árvore.

Uma representação gráfica da árvore de decisão gerada pode ser vista na seguinte imagem:



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 imagem a seguir:



Representação da árvore de decisão com profundidade 3.

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

Algoritmos não supervisionados

Não supervisionado — agrupamento

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

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.

No próximo exemplo (script agrupamento.py), utilizaremos 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, começaremos 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 `caracteristicas`. 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, treinaremos 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 conterão 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. Veja o código a seguir:

Script Agrupamento.py

```
1 import matplotlib.pyplot as plt
2 from mpl_toolkits.mplot3d import Axes3D
3 from sklearn.cluster import KMeans
4 from sklearn.datasets import load_iris
5
6 ##### Pré-processamento #####
7 # Coleta e Integração
8 iris = load_iris()
9
10 caracteristicas = iris.data
11
12 ##### Mineração #####
13 grupos = KMeans(n_clusters=3)
14 grupos.fit(X=caracteristicas)
```

A partir da linha 18, 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.

Observe os gráficos a seguir. O gráfico A contém os resultados do agrupamento e o gráfico B contém os resultados reais da amostra:

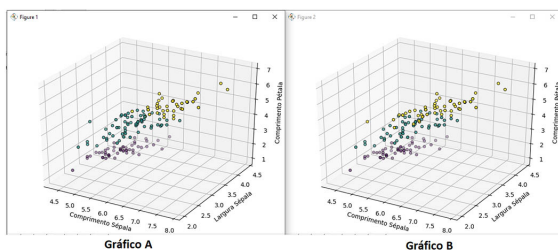


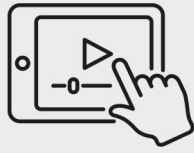
Gráfico: Resultado do agrupamento e Resultados reais da amostra.



Aplicando Python para o tratamento de dados

Neste vídeo, abordaremos o treinamento supervisionado.

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



Exercícios de aplicação do Python para o tratamento de dados

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



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 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?

- I. Coleta e Integração.
- II. Codificação.
- III. Construção de atributos.
- IV. 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

E I, II e IV

Parabéns! A alternativa B está correta.

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

Questão 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

E Limpeza dos dados

Parabéns! A alternativa B está correta.

A codificação categórico-numérica transforma string em números.

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.



Podcast

Ouçá este podcast sobre os principais assuntos abordados no tema.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.



Referências

CONCORRENTE. Dicionário On-line de Português –2009-2020. Consultado na Internet em: 25 mar. 2022.

FAYYAD, U. M.; PIATETSKY-SHAPIO, 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.

MATPLOTLIB. **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.