Linguagem de Programação

Python é uma linguagem de programação orientada a objetos. Sua essência foi criada para ter comandos e estruturas simples, de fácil leitura e compreensão. A seguir, conheça as principais características dessa linguagem.

Agora que você já conheceu um pouco mais sobre a ferramenta Python vamos aprofundar nessa linguagem de programação.

Variáveis e Tipos Básicos de Dados em Python

A construção de um algoritmo envolve entrada, processamento e saída. Para que o processamento ocorra, é necessário um meio de armazenar valores temporariamente, razão pela qual surge o conceito de variáveis. Uma variável é um espaço alocado na memória RAM.

Muitas linguagem de programação possuem variáveis com tipo primitivo, como *int* ou *string* – em Python esses tipos primitivos não existem.

Na linguagem Python, o tipo de variável é identificado no momento que se atribui um valor a ela. Portanto, uma variável refere-se a um valor.

Na verdade, em Python não há a necessidade de definir estaticamente o tipo de variável, como em outras linguagens de programação.

Banin (2018) explica que “O modelo de dados do Python (Python Data Model, em inglês) adota como paradigma que todo dado em um programa escrito com Python é representado por um objeto”.

Todo objeto Python tem três aspectos:

- Um identificador.
- Um tipo.
- Um conteúdo.

Os códigos a seguir exibem o resultado para três objetos: int, str e float (tipos de variáveis, sem declaração de tipo.

```
# String:
print("Olá")

# Inteiro:
print(10)

# Float:
print(3.141592)
```

Exemplos de códigos que atribuem valores à variável.

```
valor = 10
print(type(valor))
```

```
valor = "nome"
print(type(valor))

valor = 10.5
print(type(valor))
```

Em outras linguagens de programação, tais objetos seriam variáveis primitivas. Por serem objetos, tais tipos possuem métodos e comportamentos que outras linguagens não suportam – por exemplo, multiplicar uma string por um valor inteiro.

Operadores Numéricos

Como já destacamos, todo objeto em Python possui um identificador (o nome), um tipo e o conteúdo. Diferentes tipos de objetos vão suportar diferentes operações. Cada uma destas deve ser escolhida de acordo com o problema a ser resolvido.

Os tipos numéricos, naturalmente, suportam operações matemáticas entre si, devendo ser respeitada a ordem de precedência dos operadores:

1. Primeiro resolvem-se os parênteses, do mais interno para o mais externo.
2. Exponenciação.
3. Multiplicação e divisão.
4. Soma e subtração.

Exemplos de Operações Matemáticas

Se a ordem de precedência não for respeitada, o resultado pode ser equivocado.

Estruturas Condicionais e Repetição e Lógicas em Python

Em geral, em um programa você tem opções de caminhos ou lista de comandos que nada mais são que trechos de código que podem ser executados, devendo-se tomar decisões sobre qual trecho de código será executado em um determinado momento. Em Python estruturas são bem definidas, como podemos observar a seguir.

Estruturas Condicionais em Python

Em Python, o comando para a estrutura condicional é dado por *if*, *else* e *elif*.

- **If (se):** Estrutura de condição que avalia uma expressão executando caso essa seja verdadeira.
- **Else (senão):** Instrução dependente que completa a estrutura *if* executada quando expressão não ser satisfatória (falsa).
- **Elif (senão se):** Abreviação do else if executa expressões intermediárias.

Estrutura de Repetição em Python

Essas estruturas são criadas para a necessidade de se executar várias vezes o mesmo trecho de código.

- **While (enquanto):** Permite que um conjunto de instruções seja executada enquanto uma condição for verdadeira.
- **For (para):** Executa uma instrução por uma certa quantidade de vezes.

Estruturas lógicas em Python

Em Python utilizamos operadores booleanos para construir estruturas de decisões mais complexas.

- **And (e):** Retorna verdadeiro quando ambos A e B forem verdadeiros.

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

```
a = False
b = True
print(f"Resultado de \"A and B\": {a and b}")
```

- **Or (ou):** Retorna verdadeiro se A ou B ou ambos forem verdadeiros.

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

```
a = False
b = True
print(f"Resultado de \"A and B\": {a or b}")
```

- **Not (não):** Retorna verdadeiro se A for falso. Ou retorna falso, quando A for verdadeiro.

```
a = False
print(f"Resultado de \"A\": {not a}")
```

Agora que você já está familiarizado com algumas estruturas em Python, vamos aprofundar nesse tema.

Estruturas Condicionais em Python

Uma estrutura condicional verifica a condição dos argumentos passados e executa um comando caso a condição seja verdadeira. Para a construção de estruturas condicionais em Python são utilizados os comandos:

- **If:** Por padrão, o bloco de instrução que estiver abaixo da instrução *if* será executado quando a expressão contida na estrutura *if* for verdadeira.
- **Else:** Instrução complementar da estrutura *if*, deve ser executada quando a expressão definida for igual a falso.
- **Elif:** Trata-se de uma abreviação do eles *if* usado para fazer as condições intermediárias.

Além dos comandos, o bloco condicional é marcado pelos dois-pontos ao final da condição e pela identação do bloco de comandos a ser executado caso a condição seja verdadeira.

Exemplos:

```
codigo = 5222

if codigo == 5222:
    print("Compra à vista!")
elif codigo == 5333:
    print("Compra à prazo no boleto!")
elif codigo == 5444:
    print("Compra à prazo no cartão!")
else:
    print("Código não cadastrado!")
```

No exemplo observamos que o interpretador Python identifica a relação de subordinação entre estruturas condicionais e blocos de comando pelo recuo que há na digitação das linhas do programa.

- Na linha 1 temos a condição inicial.
- Na linha 2 um exemplo de linhas subordinadas estão digitadas com alguns espaços em branco à esquerda. Isso recebe o nome de identação.
- Na linha 3, temos o primeiro *elif*, que está no mesmo nível de identação do *if*, pois um não está subordinado a outro. O mesmo acontece para o *elif* da linha 5 e o *else* da linha 7.

Em Python, a identação é obrigatória sempre que houver um ou mais comandos subordinados a outro.

“Generalizando, em Python, todo conjunto de comandos subordinados deve estar identado em relação ao seu comando proprietário. Isso vale para *if-else*, *while*, *for*, *try*, *def* e qualquer outro em que exista a relação de subordinação.”

Praticando

Imagine que você deseja acessar pela primeira vez algum sistema protegido por senha. Nesse exercício vamos implementar um simples verificador de senha do usuário e para isso utilizaremos as estruturas condicionais para permitir o login de um usuário. Vamos aplicar apenas as estruturas *if-else*.

Observe a sintaxe para esse comando:

```
login = input("Digite o seu login: ")
senha = input("Digite sua senha: ")

if login == "user" and senha == "teste123":
    print(f"Bem-Vindo, {login}.")
else:
    print("Login falhou, verifique seus dados e tente novamente!")
```

Nas linhas 1 e 2, *input* é a função que requisita os dados de login e senha do usuário. A estrutura condicional *if-else*, linhas 3 e 4, determinam se a condição será satisfeita e qual linha será executada. Na linha 3, é testado se o login e a senha do usuário são iguais a “user” e “teste123”.

E se tivermos vários usuários para realizar a verificação da senha, como seria? Podemos utilizar a estrutura elif, para checar múltiplas condições e executar determinadas linhas de código.

Observe o código para verificação de senha de 4 usuários cadastrados no sistema:

```
login = input("Digite o seu login: ")
senha = input("Digite sua senha: ")

if login == "user" and senha == "teste123":
    print(f"Bem-Vindo, {login}.")
elif login == "admin" and senha == "123":
    print(f"Bem-Vindo, {login}.")
elif login == "root" and senha == "toor":
    print(f"Bem-Vindo, {login}.")
elif login == "adm" and senha == "ad123":
    print(f"Bem-Vindo, {login}.")
else:
    print("Login falhou, verifique seus dados e tente novamente!")
```

As linhas 5, 7 e 9 utilizamos a estrutura *elif* para verificar outras senhas de usuário, Na linha 11, temos a estrutura eles, que só será acionada se a entrada das linhas 1 e 2 for diferente dos logins e senhas permitidos.

Estruturas de Repetição

O comando For em Python

A instrução *for* em Python faz uma iteração sobre os itens de qualquer sequência – por exemplo, iterar sobre os caracteres de uma palavra, já que esta é um tipo de sequência.

O bloco de comandos a ser executado deve ser identado em razão da relação e da subordinação.

A sintaxe do comando envolve:

- **For:** A palavra reservada for seguido de uma variável de controle.
- **In:** A palavra reservada in.
- **::** E uma sequência acompanhada por dois-pontos.

O comando While em Python

O comando *while* deve ser utilizado para construir e controlar a estrutura decisão, sempre que o número de repetições não seja conhecido. Por exemplo, quando queremos solicitar e testar se o número digitado pelo usuário é par ou ímpar. Quando ele digitar zero, o programa se encerra. Veja, não sabemos quando o usuário irá digitar, portanto somente o *while* é capaz de solucionar esse problema.

A construção de uma sequência numérica pode ser feita com o comando *range()*. Ao consultar a documentação oficial, encontramos *range()* na lista de funções built-it em Python. Veja o exemplo seguinte:

```
print(tuple(range(10))) # Sequência de 0 a 9

print(tuple(range(1, 11))) # Sequência de 1 a 11

print(tuple(range(0, 30, 5))) # Sequência de 0 a 30 de 5 em 5

print(tuple(range(0, -10, -1))) # Sequência negativa de 0 a -10

print(tuple(range(0))) # Tupla vazia
```

Banin (2018), traz uma elucidação importante sobre o tipo range, apresentando-o como uma expressão, e não como uma função.

Em Python, quando a mesma instrução precisa ser executada várias vezes seguidas podemos utilizar loop. Esse comando permite executar um bloco de código repetidas vezes, até que uma dada condição seja atendida.

Praticando

Agora iremos colocar em prática a codificação do loop em meio dos comandos *for* e *while*.

O loop *for* é muito utilizado em conjunto com o range, o loop facilita a iteração dos valores sem a necessidade de escrever código para alterar esse valor, dificultando a ocorrência de um loop infinito. Observe a sintaxe do exemplo que a contagem inicia em 1 e para em 10.

```
for contagem in range(1, 10):
    print(contagem)
```

A estrutura de repetição *while* executa um conjunto de instruções enquanto uma condição for verdadeira. O loop é interrompido quando a condição passa a ser falsa.

Observe o exemplo a seguir, o loop *while* continuará executando as duas linhas de código enquanto a contagem for menor que 10, no momento em que atingir 10, a execução é finalizada.

```
contagem = 0

while contagem < 10:
    print(contagem)
    contagem += 1
```

Estruturas Lógicas em Python: And, Or, Not

Em Python utilizamos operadores booleanos para construir estruturas de decisões mais complexas. Nesses operadores o verdadeiro é chamado de *true* (igual a 1) e o falso é chamado de *false* (igual a 0).

- **And:** Faz a expressão lógica E. Dessa forma esse operador retorna um valor verdadeiro somente se as duas expressões forem verdadeiras.
- **Or:** Faz a expressão lógica OU. Dessa forma esse operador retorna um valor falso somente se as duas expressões forem falsas.
- **Not:** Faz a expressão lógica NÃO. Esse operador muda o valor de seu argumento, ou seja, se o argumento for verdadeiro, a operação o transformará em falso e vice-versa.

Os tipos de estruturas aqui apresentados, condicionais e de repetição, representam uma parte fundamental da linguagem de programação em Python, sendo assim, é muito importante conhecer a sintaxe e o funcionamento dessas estruturas.

Funções em Python

Uma função é uma forma de organização, usada para delimitar ou determinar quais tarefas podem ser realizadas por uma determinada divisão. Em linguagem de programação, uma função é uma sequência de instruções que processa um ou mais resultados que são chamados de parâmetros. Em Python temos as funções built-in e as funções definidas pelo usuário.

Funções built-in em Python

Uma função built-in é um objeto que está integrado ao núcleo do interpretador Python. Ou seja, não precisa ser feita nenhuma instalação adicional, já está pronto para uso.

O interpretador Python possui várias funções disponíveis, conforme podemos conservar na tabela de funções built-in em Python.

Veja a descrição de algumas dessas funções:

- *print():* Usada para imprimir um valor na tela.
- *enumarate():* Usada para retornar a posição de um valor em uma sequência.
- *input():* Usada para capturar um valor digitado no teclado.
- *int()* e *float():* Usadas para converter um valor no tipo inteiro ou float.
- *type():* Usada para saber qual é o tipo de um objeto (variável).

Funções Definidas pelo Usuário

Além das funções built-in, muitas vezes as soluções exigem a implementação de funções específicas, as quais são chamadas de funções definidas pelo usuário.

Funções, também conhecidas como subprogramas ou sub-rotinas, são pequenos blocos de código aos quais se dá um nome, desenvolvidos para resolver tarefas específicas. Tais funções constituem um elemento de fundamental importância na moderna programação de computadores, a ponto de ser possível afirmar que atualmente nenhum programa de computador é desenvolvido sem o uso desse recurso.

A sintaxe de uma função em Python é feita com:

- A palavra reservada *def*.
- O nome da função.
- Os parênteses que indicam se existem ou não parâmetros para a função.
- E o comando *return* (que é opcional).

A seguir, veja que existem três sintaxes diferentes:

- Uma função que não recebe e não retorna valores.
- Uma função que não recebe argumento, mas retorna valores.
- Uma função que recebe argumento e retorna valor.

Funções Anônimas em Python

Expressões lambda, (às vezes chamadas de formas lambda) são usadas para criar funções anônimas. Uma função anônima é uma função que não é construída com o "*def*" e que, por isso, não possui nome. Esse tipo de construção é útil quando a função faz somente uma ação e é usada uma única vez.

Os argumentos de uma função podem ser posicionais ou nominais. No primeiro, cada argumento será acessado pela sua posição, no segundo, pelo nome. Além disso, os parâmetros podem ter valores padrões, o que torna sua passagem não obrigatória.

Listas, Tuplas, Set, Dicionário em Python

Em Python, é permitido a utilização de vários tipos de estruturas de dados. As principais estruturas são as listas, tuplas, set e dicionário.

Listas

Lista é uma estrutura de dados do tipo sequencial que possui como principal característica ser mutável. Ou seja, novos valores podem ser adicionados ou removidos da sequência.

Na lista, os dados são posicionais e sequenciais.

Observe que o primeiro valor ocupa a posição zero da lista, já o último ocupa a posição n – 1, em que n é a quantidade de elementos que a lista é capaz de armazenar.

Como Criar uma Lista em Python

Em Python, uma das maneiras de criar uma lista é colocando os valores entre colchetes, conforme código a seguir:

```
vogais = ["A", "E", "I", "O", "U"]
```

A lista pode ser criada sem nenhum elemento, e a inserção pode ser feita posteriormente:

```
vogais = []

vogais.append("A")
vogais.append("E")
vogais.append("I")
vogais.append("O")
vogais.append("U")
```

Para acessar o valor guardado em uma lista, basta indicar o nome da variável e, entre colchetes, a posição do elemento, ou a fatia (slice) de valores que se deseja:

```
print(vogais[3])
print(vogais[3:])
```

List Comprehension

Uma maneira muito elegante de criar uma lista´e usando a list comprehension. Também chamada de listcomp, é uma forma pythônica de criar uma lista com uso de um objeto iterável.

Esse tipo de técnica é utilizado quando, dada uma sequência, deseja-se criar uma nova sequência, porém com as informações originais transformadas ou filtradas por um critério.

Os comandos for-in são obrigatórios.

As variáveis item e lista dependem do nome dado no programa. Veja um exemplo de sintaxe utilizando a listcomp:

```
print([2 * x for x in range(10)])
```

Tuplas

As tuplas também são estruturas de dados do grupo de objetos do tipo sequência.

A grande diferença entre listas e tuplas é que, com as primeiras, uma vez que são mutáveis, conseguimos fazer atribuições a posições específicas (por exemplo, *lista[2] = “Maçã”*), o que, nas segundas, não é possível, pois estas são objetos imutáveis.

Em Python, uma das maneiras de criar uma tupla é colocando os valores entre parênteses, conforme código a seguir:

```
vogais = ("A", "E", "I", "O", "U")
```

Sets

Um objeto do tipo set habilita operações matemáticas de conjuntos, tais como: União, intersecção, diferença, etc. Esse tipo de estrutura pode ser usado em testes de associação e remoção de valores duplicados de uma sequência.

Em Python, uma das maneiras de se criar um objeto do tipo set é colocando os valores entre chaves, conforme código a seguir:

```
vogais = {"A", "E", "I", "O", "U"}
```

Dicionários

As estruturas de dados que possuem um mapeamento entre uma chave e um valor são consideradas objetos do tipo mapping. Em Python, o objeto que possui essa propriedade é o dict (dicionário). Tal objeto é mutável, ou seja, com ele conseguimos atribuir um novo valor a uma chave já existente.

Em Python, uma das maneiras de criar um objeto do tipo dicionário é colocando as chaves e os valores entre estas, conforme código a seguir:

```
cadastro = {"nome": "João", "idade": 30, "cidade": "São Paulo"}
```

Para acessar um valor em um dicionário, basta digitar:

```
print(cadastro["nome"])
```

E para atribuir um novo valor, use:

```
cadastro["nome"] = "Sérgio"
```

Dicionários são estruturas de dados não sequenciais que permitem o acesso a um valor por meio de uma chave. O tipo dicionário apresenta diversos métodos.

Algoritmos de Busca

Os algoritmos computacionais são desenvolvidos e usados para resolver os mais diversos problemas. Algoritmos de busca, nesse universo, como o nome sugere, resolvem problemas relacionados ao encontro de valores em uma estrutura de dados.

Diversas são as aplicações que utilizam esse mecanismo.

Você já fez alguma pesquisa no Google hoje? Já utilizou seu app do banco? Já procurou alguém nas redes sociais?

Todas essas aplicações utilizam mecanismos de busca. Um grande diferencial entre uma ferramenta e outra é a velocidade da resposta: Quanto mais rápido ela for, mais gostamos dela.

Por trás de qualquer tipo de pesquisa existe um algoritmo de busca implementado.

Nesta aula vamos aprender sobre os algoritmos computacionais.

Busca Linear ou Sequencial

A busca começa por uma das extremidades da sequência e vai percorrendo-a até encontrar (ou não) o valor desejado. Fica claro que uma pesquisa linear examina todos os itens da sequência até encontrar o item de destino, o que pode ser muito custoso computacionalmente.

Código	Explicação
def	Definição da função.

executarBuscaLinear(lista, valor):	
tamanhoLista = len(lista)	Definição do tamanho da lista.
tamanhoLista = len(lista)	Estrutura de repetição para percorrer toda a lista.
if valor == lista[i]:	Condição para verificar se é o valor.
return True	Retorno, caso encontre o valor.
return False	Retorno, caso não encontre o valor.

Algoritmos de Busca Sequencial

Para procurar elementos em uma lista por meio de algoritmo de busca sequencial, é necessário percorrer todos os elementos da lista até encontrar o elemento procurado. Para isso, é realizada uma comparação do valor do elemento que se deseja encontrar na lista com o valor de cada posição na lista.

Mesmo quando um dos elementos não estiver na lista, todos eles devem ser visitados. Os elemen tos da lista podem já estar dispostos em uma sequência ordenada, ou não. Para cada um dos casos, haverá um comportamento diferenciado em termos de tempo de execução do algoritmo.

No exemplo de algoritmo de busca sequencial que considera um vetor de entrada com números desordenados. Temos uma estrutura de repetição “while”:

```
def buscaSequencial(lista, elemento):
    pos = 0
    encontrado = False

    while pos < len(lista) and not encontrado:
        if lista[pos] == elemento:
            encontrado = True
        else:
            pos += 1

    return encontrado

testeLista = [2, 10, 8, 15, 18, 20, 12, 1]

print(buscaSequencial(testeLista, 5))
print(buscaSequencial(testeLista, 15))
```

Analisando a Estrutura de Busca Sequencial:

- **Linha 5:** A estrutura de repetição “while” permitirá percorrer a lista e comparar o elemento procurado com o elemento que está na posição da lista, nesse caso, enquanto a posição for menor que o tamanho da lista e o elemento não for encontrado.
- **Linha 6:** Na linha 6 do código, a estrutura condicional “if”, “else” traz a condição para encontrar ou não o elemento do vetor – no caso, se a posição da lista corresponde ao elemento procurado (valor), será exibido True (verdadeiro), se não corresponder, haverá um incremento e o próximo elemento será visitado na sequência da lista.

- **Linha 13:** Na linha 13 é mostrado um vetor com 8 elementos (0 a 7 elementos). E na linha 14, é buscado o elemento 5 na lista. Como o elemento não está na lista, observe que todas as posições serão percorridas, razão pela qual todos os elementos da lista serão visitados.
- **Linha 14:** Na linha 14, temos um teste em que se busca o elemento de valor 15. Nesse caso, quatro elementos da lista serão percorridos até que se encontre o valor 15.

Busca Sequencial em um Vetor Ordenado

E, se a lista já tiver ordenada, como é realizada a busca do elemento procurado?

Muito bem, já conhecemos como se processa uma busca sequencial que considera um vetor de entrada com números desordenados. Mas e, se a lista já estiver ordenada, como é realizada a busca do elemento procurado?

Observe a seguinte estrutura:

```
def buscaSequencialOrdenada(lista, elemento):
    pos = 0
    encontrado = False
    para = False

    while pos < len(lista) and not encontrado and not para:
        if lista[pos] == elemento:
            encontrado = True
        else:
            if lista[pos] > elemento:
                para = True
            else:
                pos += 1

    return encontrado

testeLista = [1, 2, 8, 10, 13, 15, 18, 20]

print(buscaSequencialOrdenada(testeLista, 5))
print(buscaSequencialOrdenada(testeLista, 15))
```

Algoritmo de Busca Binária

Outro algoritmo usado para buscar um valor em uma sequência é o de busca binária.

A primeira grande diferença entre o algoritmo de busca linear e o algoritmo de busca binária é que neste os valores precisam estar ordenados.

A lógica é a seguinte:

1. Encontra o item no meio da sequência.
2. Se o valor procurado for igual ao item do meio, a busca encerra.
3. Se não, verifica se o valor buscado é maior ou menor que o valor central.
4. Se for maior, então a busca acontecerá na metade superior da sequência (a inferior é descartada), se não for maior, a busca acontecerá na metade inferior da sequência (a superior é descartada).
5. Repete os passos: 1, 2, 3, 4.

Atenção: Veja que o algoritmo, ao encontrar o valor central de uma sequência, a divide em duas partes, o que justifica o nome de busca binária.

Vamos ilustrar o funcionamento do algoritmo, na busca pelo número 14 em uma certa sequência numérica.

Veja que o algoritmo começa encontrando o valor central, o qual chamamos de m1. Como o valor buscado não é o central, e é maior que o elemento do meio da lista, a busca, então, passa a acontecer na metade superior. Dado o novo conjunto, novamente é localizado o valor central, o qual chamamos de m2, que também é diferente e menor do que o valor buscado. Mais uma vez a metade superior é considerada e, ao localizar o valor central (m3), agora sim o valor procurado, então o algoritmo encerra.

O código que representa a figura apresentada será escrito da seguinte maneira:

```
def buscaBinaria(lista, elemento):
    minimo = 0
    maximo = len(lista) - 1
    encontrado = False

    while minimo <= maximo and not encontrado:
        meioLista = (minimo + maximo) // 2

        if lista[meioLista] == elemento:
            encontrado = True
        else:
            if elemento < lista[meioLista]:
                maximo = meioLista - 1
            else:
                minimo = meioLista + 1

    return encontrado

testeLista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
print(buscaBinaria(testeLista, 14))
```

Vamos a outro exemplo de aplicação do algoritmo de busca binária com utilização do mesmo vetor ordenado apresentado para a busca sequencial.

O código para esse exemplo será escrito da seguinte maneira:

```
def buscaBinaria(lista, elemento):
    minimo = 0
    maximo = len(lista) - 1
    encontrado = False

    while minimo <= maximo and not encontrado:
        meioLista = (minimo + maximo) // 2

        if lista[meioLista] == elemento:
            encontrado = True
        else:
            if elemento < lista[meioLista]:
                maximo = meioLista - 1
            else:
```

```
                minimo = meioLista + 1

    return encontrado

testeLista = [1, 2, 8, 10, 13, 15, 18, 20]
print(buscaBinaria(testeLista, 5))
print(buscaBinaria(testeLista, 15))
```

Agora vamos analisar alguns pontos importantes desse exemplo de aplicação de busca:

- **Linha 6:** Na linha 6, temos a estrutura de repetição “while”, que será executada enquanto o primeiro elemento da lista (mínimo) for menor ou igual ao máximo (último elemento) e o elemento procurado não for encontrado.
- **Linha 7:** Na linha 7, identificamos o índice associado à metade da lista.
- **Linha 8:** Na linha 8, temos uma estrutura de condição “if” em razão da qual, basicamente, verifica-se que, se o elemento do meio da lista for o valor procurado, será retornado o True (linha 9).
- **Linha 10:** Na linha 10, temos a condição “else”, que verifica se o elemento procurado é menor que o valor do meio da lista. Se for maior, então a busca acontecerá na metade superior da sequência (a inferior é descartada), se não for, a busca acontecerá na metade inferior da sequência (a superior é descartada).

Algoritmos de busca compõem o arsenal de algoritmos tradicionais da computação. Quando é conhecida a sequência de passos, ou seja, o pseudocódigo, basta escolher uma linguagem de programação e implementá-la.

Todo algoritmo deve ter uma medida de complexidade – ou seja, a quantidade de recurso computacional é necessária para executar tal solução. A análise da complexidade do algoritmo fornece mecanismos para medir o desempenho de um algoritmo em termos de “tamanho do problema versus tempo de execução” (Toscani, Veloso, 2012). A análise da complexidade é feia em duas dimensões: Espaço e tempo. Embora ambas as dimensões influenciem na eficiência de um algoritmo, o tempo que ele leva para executar é tido como a característica mais relevante.

Algoritmos de Ordenação

Os algoritmos de ordenação determinam uma forma de organizar os dados em uma determinada sequência. Em geral, pode-se ordenar os elementos nas ordens numérica ou alfabética. Existem diversas aplicações do dia a dia, por exemplo, os objetos podem ser ordenados por tamanho, altura, peso, cor, nomes, em ordem alfabética, entre outras opções.

Como uma aplicação do cotidiano, imagine que você vai utilizar algum sistema de compra online e deseja listar os itens por ordem de preço (do mais barato para o mais caro). Essa é uma simples aplicação de ordenação utilizada na rotina diária.

Nessa aula, vamos ver como ocorre o funcionamento dos algoritmos de ordenação por meio de animações.

Classificação dos Algoritmos

Antes de explorarmos o funcionamento dos algoritmos de ordenação, é importante conhecer um pouco sobre a classificação deles. Os algoritmos de ordenação podem ser classificados quanto à complexidade da seguinte forma:

- **Complexidade O (N2):** Nesse grupo estão os algoritmos selection sort, bubble sort e insertion sort. Esses algoritmos são lentos para ordenação de grandes listas, poré são mais intuitivos de entender e possuem codificação mais simples.
- **Complexidade O (N log N):** Deste grupo, vamos estudar os algoritmos merge sort e quick sort. Tais algoritmos possuem performance superior, porém são um pouco mais complexos de serem implementados.

Algoritmo de Ordenação por Inserção (Insertion Sort)

O algoritmo de ordenação por inserção é iniciado a partir do segundo valor no vetor, pois o primeiro elemento será uma referência para a ordenação. Ou seja, o segundo elemento do vetor será comparado com o primeiro.

O vetor é percorrido da esquerda para a direita. Nesse caminho, compara-se sempre o elemento da direita com os elementos à esquerda de modo que os elementos mais à esquerda sejam organizados e ordenados. O algoritmo de ordenação por inserção tem funcionamento similar ao das pessoas que ordenam cartas em um jogo de baralho.

A seguir, podemos verificar e testar o algoritmo em Python:

```python
def executarInsertionSort(lista):
    n = len(lista)

    for i in range(1, n):
        valorInserir = lista[i]
        j = i - 1

        while j >= 0 and lista[j] > valorInserir:
            lista[j + 1] = lista[j]
            j -= 1

        lista[j + 1] = valorInserir

    return lista

lista = [10, 8, 7, 3, 2, 1]
print(executarInsertionSort(lista))
```

Algoritmo de Ordenação por Seleção (Selection Sort)

Seleção pelo Valor Mínimo

O princípio de funcionamento deste algoritmo é transferir o menor valor do vetor para a primeira posição e, em seguida, o segundo menor valor para a segunda posição, e assim, sucessivamente, para os n - 1 elementos até os últimos dois elementos.

Agora podemos verificar e testar o algoritmo em Python:

```
def executarSelectionSort(lista):
    n = len(lista)

    for i in range(0, n - 1):
        min = i

        for j in range(i + 1, n):
            if lista[j] < lista[min]:
                min = j

        lista[i], lista[min] = lista[min], lista[i]

    return lista


lista = [10, 8, 7, 3, 2, 1]
print(executarSelectionSort(lista))
```

Algoritmo de Ordenação por Bolha (Bubble Sort)

O algoritmo de ordenação por bolha (ou flutuação) consiste naquele baseado em comparações, em que se percorre vetor ou lista múltiplas vezes. A cada passagem, os pares de elementos adjacentes do vetor são comparados e, depois disso, são trocados se não estiverem ordenados. Esse processo segue para todos os elementos não ordenados da lista.

É importante destacar que esse tipo de algoritmo não é indicado para uma grande quantidade de dados.

Agora podemos verificar e testar o algoritmo em Python:

```
def executarBubbleSort(lista):
    n = len(lista)

    for i in range(n - 1):
        for j in range(n - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]

    return lista

lista = [10, 8, 7, 3, 2, 1]
print(executarBubbleSort(lista))
```

Algoritmo de Ordenação Merge Sort

O algoritmo de ordenação merge sort é baseado no princípio dividir para conquistar. Basicamente, o vetor é dividido em duas metades e, em seguida, novamente é dividido em outras duas partes – ao repetir a divisão do vetor, em um determinado momento teremos vários vetores com apenas um elemento.

Cada metade é ordenada recursivamente, bem como o são as partes com um elemento. Assim, realizamos a junção (merge) dos subvetores ordenados, iniciando a junção de dois a dois de todos os vetores de um elemento e gerando vetores de tamanho dois. Depois disso, o processo é repetirdo para vetores de tamanho três, quatro, e assim sucessivamente.

Agora você pode testar o algoritmo:

```
def executarMergeSort(lista):
    if len(lista) <= 1:
        return lista
    else:
        meio = len(lista) // 2
        esquerda = executarMergeSort(lista[:meio])
        direita = executarMergeSort(lista[meio:])

        return executarMerge(esquerda, direita)

def executarMerge(esquerda, direita):
    subListaOrdenada = []
    topoEsquerda, topoDireita = 0, 0

    while topoEsquerda < len(esquerda) and topoDireita < len(direita):
        if esquerda[topoEsquerda] <= direita[topoDireita]:
            subListaOrdenada.append(esquerda[topoEsquerda])
            topoEsquerda += 1
        else:
            subListaOrdenada.append(direita[topoDireita])
            topoDireita += 1

    subListaOrdenada += esquerda[topoEsquerda:]
    subListaOrdenada += direita[topoDireita:]

    return subListaOrdenada

lista = [10, 8, 7, 3, 2, 1]
print(executarMergeSort(lista))
```

Algoritmo de Ordenação Quick Sort

Nos algoritmos quick sort, primeiramente escolhemos o pivô, que corresponde ao elemento do meio do vetor. Depois, nós trocamos os elementos no início do vetor com os elementos do final – até no início do vetor serão apenas esses elementos, os quais são menores ou iguais ao pivô, e no final do vetor, este permanecerá apenas com esses elementos, que são menores ou iguais ao pivô. Por fim, se no início ou no fim há mais que um elemento, repetimos todo o processo para essas partes (elementos).

A seguir podemos verificar e testar o algoritmo no emulador implementado em Python:

```
def executarQuickSort(lista, inicio, fim):
    if inicio < fim:
        pivo = executarParticao(lista, inicio, fim)

        executarQuickSort(lista, inicio, pivo - 1)
        executarQuickSort(lista, pivo + 1, fim)

    return lista

def executarParticao(lista, inicio, fim):
    pivo = lista[fim]
    esquerda = inicio

    for direita in range(inicio, fim):
        if lista[direita] <= pivo:
```

```
            lista[direita], lista[esquerda] = lista[esquerda], lista[direita]
            esquerda += 1

    lista[esquerda], lista[fim] = lista[fim], lista[esquerda]

    return esquerda

lista = [10, 8, 7, 3, 2, 1]
print(executarQuickSort(lista, inicio = 0, fim = len(lista) - 1))
```

Nesta aula você conheceu os algoritmos de ordenação, os quais já pode testar na prática. Além de ser útil trabalhar com dados ordenados, estudar essa classe de algoritmos permite explorar várias técnicas de programação, tais como a recursão e o problema de dividir para conquistar. Com esse conhecimento, podemos arriscar dizer que você, desenvolvedor, passar de um nível de conhecimento mais "usuário" para um nível mais técnico, mais profundo, o que lhe capacita a entender vários mecanismos que estão por trás de funções prontas que encontramos no dia a dia.

Classes e Métodos em Python

O desenvolvimento de software orientado a objeto (OO) existe desde o início dos anos 1960, mas foi somente em meados da década de 1990 que o paradigma orientado a objetos começou a ganhar impulso. Uma linguagem é entendida como orientada a objetos se ela aplica o conceito de abstração e suporta a implementação do encapsulamento, da herança e do polimorfismo.

Definições Importantes

Antes de aprendermos como criar uma classe em Python, vamos conhecer os conceitos de objeto, classe e instância:

- **Objetos:** São os componentes de um programa OO. Um programa que usa a tecnologia OO é basicamente uma coleção de objetos.
- **Classe:** Uma classe é um modelo para um objeto. Segundo a Python Software Fundation (PSF, 2020), podemos considerar uma classe como uma forma de organizar os dados (de um objeto) e seus comportamentos.
- **Instância:** Entende-se por instância, a existência física, em memória, do objeto.

Vamos pensar na construção de uma casa: Antes do "objeto casa" existir, um arquiteto fez a planta dela, determinando tudo que deveria fazer parte daquele objeto. Portanto, a classe é o modelo e o objeto é uma instância.

Como Criar uma Classe em Python

Atributos

Para criar uma classe em Python é necessária a sintaxe a seguir. Utiliza-se a palavra reservada "class" para indicar a criação de uma classe, seguida do nome e de dois pontos. No bloco indentado devem ser implementados os atributos e métodos da classe.

Exemplo de criação de uma classe:

```
class PrimeiraClasse:
    nome = None

    def imprimirMensagem(self):
        print("Olá, seja bem-vindo!")
```

Após criada uma classe, os objetos podem ser instanciados, sendo importante lembrar que uma classe determina um tipo de estrutura de dados. Os atributos e os métodos de uma classe podem ser acessados pelo objeto, colocando o nome deste seguido de ponto. Por exemplo:

```
objeto1 = PrimeiraClasse()

objeto1.nome = "Aluno 1"

print(objeto1.nome)
objeto1.imprimirMensagem()
```

Os dados armazenados em um objeto representam o estado do objeto. Na terminologia de programação OO, esses dados são chamados de atributos. Os atributos contêm as informações que diferenciam os vários objetos – neste caso, os funcionários.

Métodos

O comportamento de um objeto representa o que o objeto pode fazer. Nas linguagens procedurais, o comportamento é definido por:

- Procedimentos.
- Funções.
- Sub-rotinas.

Na terminologia de programação OO, esses comportamentos estão contidos nos métodos, e você invoca um método enviando uma mensagem para ele.

A combinação dos atributos e métodos na mesma entidade, na linguagem OO, é chamada de encapsulamento. Alguns autores também consideram como encapsulamento a prática de tornar atributos privados, encapsulando-os em métodos para guardar e acessar seus valores.

Atributos: Variáveis de Classe e de Instância

Os atributos de uma classe podem ser variáveis de instância ou da classe. Uma variável de instância significa que, para cada objeto, é guardado um valor diferente, já as variáveis de classe são comuns a todas as instâncias de uma classe. As variáveis de instâncias podem ter seus valores inicializados no momento da construção da classe, por meio do método construtor de classe.

Construtor da Classe

Ao instanciar um novo objeto, é possível determinar um estado inicial para variáveis de instância (atributos) por meio do método construtor da classe.

Em Python, o método construtor é chamado de *__init__()* e deve ser usado conforme o código a seguir. Na classe abaixo, o atributo *status*, que é uma variável de instância, recebe o valor no momento da criação do objeto, pois está no construtor:

```
class FuncionarioTecnico:
    def __init__(self, status):
        self.status = status

    nivel = "Técnico"

func1 = FuncionarioTecnico("Ativo")
func2 = FuncionarioTecnico("Licença Mestrado")

print(func1.nivel)
print(func2.nivel)
print(func1.status)
print(func2.status)
```

Todo método em uma classe deve receber como primeiro parâmetro uma variável que indica a referência à classe – por convenção, adota-se o parâmetro *self*. O parâmetro *self* será usado para acessar os atributos e métodos dentro da própria classe.

Toda variável de instância possui o prefixo *self*, pois é dessa forma que é identificado que o atributo faz parte de um objeto específico. Para se utilizar um método, dentro da classe, também é necessário utilizar o prefixo *self*.

Bibliotecas e Módulos em Python

Implementamos algoritmos, nas diversas linguagens de programação, para automatizar soluções e acrescentar recursos digitais, como interfaces gráficas e processamento em larga escala. Uma solução pode começar com algumas linhas de códigos, mas, em pouco tempo, se tornar centenas, milhares e até milhões delas. Nesse cenário, trabalhar com um único fluxo de código se torna inviável, dando origem a técnicas de implementação para organizar a solução.

Organização em Módulos

Uma opção para organizar o código é implementar funções, contexto em que cada bloco passa a ser responsável por uma determinada funcionalidade. Outra forma é utilizar a orientação a objetos e criar classes que encapsulam características e comportamentos de um determinado objeto. Conseguimos utilizar ambas as técnicas para melhorar nosso código, mas, ainda assim, estamso falando de toda a solução agrupada em um arquivo Python.

Considerando a necessidade de implementar uma solução, o mundo ideal: Modular uma solução, ou seja, fazer a separação em funções ou classes e ainda realizar a separação em vários arquivos .py.

Segundo a documentação oficial do Python, é preferível implementar, em um arquivo separado, uma funcionalidade que você possa reutilizar, criando assim um módulo.

“Um módulo é um arquivo contendo definições e instruções Python. O nome do arquivo é o nome do módulo acrescido do sufixo .py” (PSF, 2020).

Veja a seguir essa ideia, com base na qual uma solução original implementada em um único arquivo .py é transformada em três módulos que, inclusive, podem ser reaproveitados, conforme aprenderemos.

Bibliotecas

Falamos em módulo, mas em Python se ouve muito o termo biblioteca. O que será que eles têm em comum?

Na verdade, um módulo pode ser uma biblioteca de códigos! Veja a seguir que temos o módulo *math*, que possui fiversas funções matemáticas, e o módulo *os*, que possui funções de sistema operacional, como capturar o caminho, listar um diretório, criar uma nova pasta, dentre inúmeras outras.

Esses módulos são bibliotecas de funções pertinentes a um determinado assunto (matemática e sistema operacional), as quais possibilitam a reutilização de código de uma forma elegante e eficiente.

Como Utilizar um Módulo

Para utilizar um módulo, é preciso importá-lo para o arquivo, essa importação pode ser feita de maneiras distintas.

Desta forma todas as funcionalidades de um módulo são carregadas na memória.

Nesta forma de importação, somente funcionalidades específicas de um módulo são carregadas na memória. A forma de importação também determina a sintaxe para utilizar a funcionalidade. Como podemos observar no exemplo a seguir:

Exemplo 1: Importando todas as funcionalidades da biblioteca:

```
import math

print(math.sqrt(25))
print(math.log2(1024))
print(math.cos(45))
```

Exemplo 2: Importando todas as funcionalidades da biblioteca com alias:

```
import math as m

print(m.sqrt(25))
print(m.log2(1024))
print(m.cos(45))
```

Exemplo 3: Importando funcionalidades específicas de uma biblioteca:

```
from math import sqrt, log2, cos

print(sqrt(25))
print(log2(1024))
print(cos(45))
```

Classificação dos Módulos

Podemos classificar os módulos (bibliotecas) em três categorias:

- **Módulos Built-in:** São embutidos no interpretador. Ao instalar o interpretador Python, também é feita a instalação de uma biblioteca de módulos, que pode variar de um sistema operacional para outro.
- **Módulos de Terceiros:** São criados por terceiros e disponibilizados via PyPI. PyPI é a abreviação para Python Package Index, que é um repositório para programas Python. Programadores autônomos e empresas podem criar uma solução em Python e disponibilizá-la em forma de biblioteca no repositório PyPI. Dessa forma, todos usufruem e contribuem para o crescimento da linguagem.
- **Módulos Próprios:** São criados pelo desenvolvedor. Cada módulo pode importar outros módulos, tanto os pertencentes ao mesmo projeto, quanto os built-in ou de terceiros.

Aplicação de Banco de Dados com Python

Grande parte dos softwares que são desenvolvidos (senão todos) acessam algum tipo de mecanismo para armazenar dados. A persistência dos dados pode ser feita em arquivo, em um banco de dados relacional ou em um banco de dados NoSQL.

A teoria base dos bancos de dados relacional existe desde a década de 1970. Nessa abordagem os dados são persistidos em uma estrutura bidimensional, chamada de relação (que é uma tabela) e baseada na teoria dos conjuntos pertencentes à matemática.

Cada unidade de dados é conhecida como coluna, ao passo que cada unidade do grupo é conhecida como linha, tupla ou registro.

A Linguagem SQL

Para se comunicar com um banco de dados relacional, existe uma linguagem específica conhecida como SQL, que significa structured query language ou, traduzindo, linguagem de consulta estruturada. As instruções da linguagem SQL são divididas em três grupos: DDL, DML e DCL:

- **DDL:** É um acrônimo para data defintion language (linguagem de definição de dados). Fazem parte desse grupo as instruções destinadas a criar, deletar e modificar banco de dados e tabelas. Nesse módulo, vão aparecer comandos como *create*, *alter* e *drop*.
- **DML:** É um acrônimo para data manipulation language (linguagem de manipulação de dados). Fazem parte deste grupo as instruções destinadas a recuperar, atualizar, adicionar ou excluir dados em um banco de dados. Nesse módulo vão aparecer comandos como *insert*, *update* e *delete*.
- **DCL:** É um acrônimo para data control language (linguagem de controle de dados). Fazem parte deste grupo as instruções destinadas a manter a segurança adequada para o banco de dados. Nesse módulo vão aparecer comandos como *grant* e *revoke*.

Banco de Dados SQLite

O SQLite é uma biblioteca em linguagem C, que implementa um mecanismo de banco de dados SQL pequeno, rápido, independente, de alta confiabilidade e completo. Por ser leve e não precisar da instalação de um servidor é uma ótima opção para diversos cenários.

O SQLite lê e grava diretamente em arquivos de disco, ou seja, um banco de dados SQL completo com várias tabelas, índices, triggers e visualizações está contido em um único arquivo de disco.

O interpretador Python possui o módulo built-in sqlite3, que permite utilizar o mecanismo de banco de dados SQLite. Com esse módulo, é bastante simples criar um banco de dados SQLite e fazer a conexão.

A seguir, criamos um banco de dados chamado auladb. Esse comando criará um arquivo chamado auladb com extensão db, que o identifica como um arquivo de banco.

```
import sqlite3

conn = sqlite3.connect("auladb.db3")
```

O CRUD no Banco de Dados SQLite

Quando o assunto é banco de dados, um termo muito comum é o CRUD, um acrônimo para as quatro operações de DML que podemos fazer em uma tabela no banco de dados – podemos inserir informações (*create*), ler (*read*), atualizar (*update*) e apagar (*delete*).

Os passos necessários para efetuar uma das operações do CRUD são sempre os mesmos:

1. Estabelecer a conexão com um banco.
2. Criar um cursor e executar um comando.
3. Gravar a operação.
4. Fechar o cursor e a conexão.

Aplicação do CRUD no banco de dados SQLite

Agora que já conhecemos o CRUD, vamos inserir as informações para cada uma de suas operações:

Create

A variável *conn* guarda a instância de conexão com o banco de dados. Agora é preciso criar um cursor para executas as instruções e, por fim, gravar as alterações com o método *commit()*.

```
cursor = conn.cursor()

cursor.execute("""
insert into fornecedor
(nome, cnpj, cidade, estado, cep, datacad)
values
('Empresa A', '11.111.111/1111-11', 'São Paulo', 'SP', '11111-111', '2020-01-
01')
""")

conn.commit()
```

Read

Para ler os dados em uma tabela, também precisamos estabelecer uma conexão e criar um objeto cursor para executar a instrução de seleção. Ao executar a seleção, podemos usar o método *fetchall()*, para capturar todas as linhas mediante uma lista de tuplas.

```
cursor.execute("select * from fornecedor")

resultado = cursor.fetchall()

for linha in resultado:
    print(linha)
```

Update

Ao inserir um registro no banco, pode ser necessário alterar o valor de uma coluna, o que pode ser feito por meio da instrução SQL *update*.

```
cursor.execute("update fornecedor set cidade = 'Campinas' where id = 1")

conn.commit()
```

Delete

Ao inserir um registro no banco, pode ser necessário removê-lo no futuro, o que pode ser feito por meio da instrução SQL *delete*.

```
cursor.execute("delete from fornecedor where id = 1")

conn.commit()
```

Introdução a Biblioteca Pandas

Dentre as diversas bibliotecas disponíveis no repositório PyPI, Pandas é um pacote Python que fornece estruturas de dados projetadas para facilitar o trabalho com dados estruturados (tabelas) e de séries temporais.

Como uma ferramenta de alto nível, Pandas possui duas estruturas de dados que são as principais para a análise/manipulação de dados: A Series e o DataFrame:

- Uma Series é como um vetor de dados (unidimensional), capaz de armazenar diferentes tipos de dados.
- Um DataFrame é um conjunto de Series, ou, como a documentação apresenta, um contêiner para Series.

Ambas as estruturas possuem como grande característica a indexação das linhas, ou seja, cada linha possui um rótulo (nome) que a identifica, o qual pode ser uma string, um inteiro, um decimal ou uma data.

Veja que:

- Uma Series possui somente “uma coluna” de informação e seus rótulos (índices).

- Um DataFrame pode ter uma ou mais colunas e, além dos índices, também há um rótulo de identificação com o nome da coluna.

Para utilizar os recursos da biblioteca é preciso importar no projeto. Usa-se a seguinte convenção:

```
import pandas
```

Series

Para construir um objeto do tipo Series, é preciso utilizar o método *Series()* do pacote Pandas. O método possui o seguinte construtor:

```
pandas.Series(data=None, index=None, dtype=None, name=None, copy=False,
fastpath=False)
```

Como todos os parâmetros possuem valores padrão (default), o que permite instanciar um objeto de diferentes formas. Veja algumas dessas formas:

```
pandas.Series(data=5) # Cria uma Series com o valor

pandas.Series("Fulano Beltrano Sicrano".split()) # Cria Series com uma lista de
nomes
```

DataFrame

Para construir um objeto do tipo DataFrame é preciso utilizar o método *DataFrame()* do pacote Pandas. O método possui o seguinte construtor:

```
pandas.DataFrame(data=None, index=None, columns=None, dtype=None, name=None,
copy=False)
```

Veja alguns exemplos:

```
# Cria um DataFrame, de uma coluna a partir de uma lista:
pandas.DataFrame("Fulano Beltrano Sicrano".split(), columns=["nome"])
```

Outro exemplo:

```
nomes = "Fulano Beltrano Sicrano".split()
cpfs = "111.111.111-11 222.222.222-22 333.333.333-33".split()
emails = "fulano@gmail.com beltrano@gmail.com sicrano@gmail.com".split()
idades = [32, 22, 25]
dados = list(zip(nomes, cpfs, idades, emails))

pandas.DataFrame(dados, columns=["nome", "cpf", "idade", "email"])
```

Estruturas de dados são utilizadas para armazenar dados e diferentes estruturas possuem diferentes atributos e métodos. Com as estruturas de dados do Pandas não é diferente, tais objetos possuem atributos e métodos específicos. Existem atributos e métodos que extraem informações estruturais da Series ou do DataFrame, por exemplo, o atributo *shape* ou *dtypes*. Por outro lado, existem recursos que permitem transformar os dados em informações, por exemplo, as estatísticas, como o método *mean()* ou *median()*.

Introdução à Manipulação de Dados em Pandas

Dataset Titan

Nesta aula vamos utilizar um dos datasets (conjuntos de dados) clássicos para quem inicia o estudo na área de ciência de dados, o Titan. Essa base foi inicialmente lançada em um desafio do portal kaggle.com e possui 8 colunas.

Seleção de Colunas

Para seleciona uma coluna usa-se a sintaxe *meu_df[“coluna”]*:

```
df_titan["Age"]
df_titan["Survived"]
```

Para selecionar mais de uma coluna é preciso passar uma lista de colunas:

```
df_titan[["Age", "Fare"]]
df_titan[["Name", "PClass", "Fare"]]
```

Seleção de Linhas – Filtros

Um dos recursos mais utilizados por equipes das áreas de dados é a aplicação de filtros. Imagine a seguinte situação: Uma determinada pesquisa quer saber qual é a média de idade de todas as pessoas na sua sala de aula, bem como a média de idades somente das mulheres e somente dos homens. A distinção por gênero é um filtro! Esse filtro vai possibilitar comparar a idade de todos com a idade de cada grupo e entender se as mulheres ou os homens estão abaixo ou acima da média geral.

DataFrames da biblioteca Pandas possuem uma propriedade chamada Ioc. Essa propriedade permite acessar um conjunto de linhas (filtrar linhas), por meio do índice ou por um vetor booleano (vetor de True ou False):

- Ao criar uma condição booleana para os dados de uma coluna, obtém-se uma Seires de valores True ou False.
- Ao usar essa Series como parâmetro no Ioc, somente os registros que tiverem valor True são exibidos.

Exemplo: Filtrar somente os homens que estavam a bordo e guardar dentro de um novo DataFrame:

```
filtro_homem = df_titan["sex"] == "male"
df_titan_homens = df_titan.loc[filtro_homem]
```

Ou ainda, criar um novo DataFrame somente com os passageiros que sobreviveram:

```
filtro_sobreviventes = df_titan["survived"] == 1
df_titan_sobreviventes = df_titan[filtro_sobreviventes]
```

O filtro sempre é criado com base em condições sobre uma ou mais colunas.

Filtro com Operadores Relacionais e Lógicos

É possível criar filtros usando operadores relacionais e lógicos para criar condições comportas. Cada condição deve estar entre parênteses e deve ser conectado pelos operadores lógicos AND (&) e OR (|).

Por exemplo, criar um novo DataFrame contendo todos os homens que sobreviveram:

```
filtro_sobreviventes = df_titan["survived"] == 1
filtro_homem = df_titan["sex"] == "male"
df_titan_sobreviventes = df_titan[(filtro_sobreviventes) & (filtro_homem)]
```

Ou um novo DataFrame com todos os passageiros do sexo feminino que estavam na primeira ou segunda classe. Nesse caso, é preciso utilizar um parênteses extra para garantir a ordem de execução: Primeiro faz o OU entre pessoas que estavam na primeira e segunda classe e depois faz o E com pessoas do sexo feminino:

```
filtro_mulher = df_titan["sex"] == "female"
filtro_classe1 = df_titan["pclass"] == 1
filtro_classe2 = df_titan["pclass"] == 2

df_titan_mulheres_c1_c2 = df_titan.loc[(filtro_mulher) & ((filtro_classe1) |
(filtro_classe2))]
```

Visualização de Dados em Python

Biblioteca Matplotlib

Ao se falar em criação de gráficos em Python, o profissional precisa conhecer a biblioteca Matplotlib, pois diversas outras são construídas a partir desta.

A criação e grande parte da evolução dessa biblioteca se deve a John Hunter, que a desenvolveu como uma opção ao uso dos softwares GNUPlot e MatLab (Mc Greggor, 2015). Com a utilização da linguagem Python na área científica para trabalhar com dados, após a extração dos resultados, o cientista precisava criar seus gráficos nos outros softwares mencionados, o que se tornava inconveniente, motivando a criação da biblioteca em Python.

Figura e Eixo

Ao utilizar a biblioteca Matplotlib é preciso entender o conceito de figura e eixo (axes).

Ao criar uma figura estamos criando um espaço (tipo uma tela em branco) para se plotar o gráfico nesse espaço. Naturalmente, uma figura pode ter “subfiguras”, ou seja, uma figura pode ser dividida:

- Quando uma figura possui um único espaço, ela também possui um único eixo para plotagem.
- Quando uma figura possui duas divisões, ela possui dois eixos, quando possui três divisões, três eixos, e assim por diante.

Método *plt.plot(dados)*

Existem algumas opções de sintaxe para se criar figuras e eixos. Podemos deixar que a própria biblioteca gerencie para nós, dessa forma basta utilizar o método *plt.plot(dados)* que o gráfico será construído sobre uma figura e um eixo criados automaticamente.

A forma automática, embora prática, não permite que o desenvolvedor tenha controle sobre o eixo que gostaria de plotar, nesse caso o ideal é criar, explicitamente, uma figura com os eixos, usando a função *plt.subplots()*.

Para criar uma figura com um eixo, de forma explícita, usamos:

```
fig, ax = plt.Subplots(1, 1)
```

Para criar uma figura com dois eixos (1 linha, 2 colunas), de forma explícita, usamos:

```
fig, ax = plt.Subplots(1, 2)
```

Ao criar uma figura com mais de um eixo, temos que informar em qual vamos criar um gráfico. A variável “ax” criada no último exemplo, é um vetor de eixos, ou seja, podemos acessar cada eixo pelo índice, começando por 0.

Portanto, para plotar na figura mais à esquerda escolhemos *ax[0].plot()* e na da direita, *ax[1].plot()*.

Veja a seguir:

```
fig, ax = plt.Subplots(1, 1)

dados = range(s)

ax[0].plot(dados)
ax[1].plot(dados)
```

As figuras podem ser criadas sem nenhum eixo, com o comando:

```
fig = plt.Subplots()
```

Dessa forma, para se plotar teremos que usar a função *plt.subplot()*, que adiciona um *subplot* em uma figura existente.