Introdução aos Métodos Heurísticos de Otimização com Python

Marcelo Otone Aguiar
Geraldo Regis Mauri
Rodrigo Freitas Silva

Alegre, ES
CAUFES
2018

MARCELO OTONE AGUIAR
GERALDO REGIS MAURI
RODRIGO FREITAS SILVA

INTRODUÇÃO AOS MÉTODOS HEURÍSTICOS DE OTIMIZAÇÃO COM PYTHON

1º Edição

Alegre, ES
CAUFES
2018

CCENS-UFES
Centro de Ciências Exatas, Naturais e de Saúde
Universidade Federal do Espírito Santo
Alto Universitário, s/n, Guararema, Alegre, ES, Brasil
http://www.alegre.ufes.br/
ISBN: 978-85-61890-99-5
Editor: CAUFES
Março de 2018

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Setorial de Ciências Agrárias, Universidade Federal do Espírito Santo, ES, Brasil)

A282i Aguiar, Marcelo O, 1979-.
Introdução aos Métodos Heurísticos de Otimização com Python[e-book] / Marcelo Otone Aguiar; Geraldo Regis Mauri; Rodrigo Freitas Silva. - 1. ed. - Alegre, ES: CAUFES, 2018.

176 p.; 19 x 26,5 cm.

Inclui bibliografia.
Sistema requerido: Adobe Acrobat Reader.
Modo de acesso: World Wide Web:
<http://repositorio.ufes.br/>.

ISBN 978-85-61890-99-5

1. Otimização. 2. Linguagem Python. 3. Métodos Heurísticos.
4. Linguagem de programação.
1. Título

CDD 000
CDU 004.94

Lista de ilustrações

Lista de tabelas

Sumário

Prefácil

ESTE LIVRO tem por objetivo conduzir o leitor no aprendizado dos conceitos introdutórios do uso de métodos heurísticos na resolução de problemas de otimização. Quando diz-se "conceitos introdutórios", refere-se ao mínimo necessário que se deve saber de uma linguagem de programação e métodos heurísticos para iniciar uma pesquisa nessa área. Entende-se então como, "mínimo necessário", o usuário saber utilizar recursos na linguagem como: estruturas de decisão ou repetição, funções, listas e aplicar este conhecimento em otimização com algumas das principais heurísticas e meta-heurísticas.

É importante destacar que, como o objetivo deste livro é apoiar o aprendizado em métodos heurísticos com a linguagem Python, então é uma premissa que o leitor do livro já tenha passado pelo aprendizado da lógica de programação, pois este tema não é abordado neste livro.

Outro ponto importante em relação ao seu conteúdo, é que, não é o objetivo deste livro, exaurir todo conhecimento sobre a linguagem, suas possibilidades e peculiaridades, visto que, trata-se de um livro introdutório, desta forma, serão abordados os principais conceitos e deixar o caminho traçado para que o leitor possa investigar mais a fundo os temas que lhe interessar, relacionados à linguagem e aos métodos heurísticos.

As primeiras aulas, na parte I deste livro, tratam apenas dos conceitos introdutórios que habilitarão o leitor à construir programas na linguagem Python, o mínimo necessário para desenvolver as heurísticas. As últimas aulas, na parte II, irão abordar então, as heurísticas de forma aplicada, com exemplos na linguagem Python.

Esperamos que a leitura seja agradável e que este livro possa contribuir com o seu conhecimento.

Autores

Parte I

Parte I - Aprendendo a linguagem Python

O objetivo deste livro, conforme o próprio título indica, é: Introduzir o leitor aos métodos heurísticos de otimização com o uso do Python como linguagem de programação. Bem, são duas disciplinas diferentes, mas uma dependente da outra, uma disciplina é a linguagem de programação Python, assumindo é claro, que o leitor já tenha conhecimento de lógica de programação, outra disciplina são os métodos heurísticos aplicados na área de otimização, que no caso, em se tratando de métodos heurísticos, há uma dependência clara de uma linguagem de programação.

Assim, este livro é separado em duas partes, a parte I tem o objetivo de conduzir o leitor ao estudo da linguagem Python, apenas o mínimo necessária para programar métodos heurísticos usando apenas a técnica de programação estruturada. Sim, sabemos que a linguagem Python é orientada a objetos! Contudo, a experiência mostra que as heurísticas escritas de forma estruturada, têm melhor performance que as heurísticas que adotam técnicas de orientação a objetos, assim, nem mesmo será tratada a **OO** neste livro, se for utilizada em algum momento, será pura e simplesmente pela característica nativa da linguagem.

Assim sendo, a parte I aborda os conceitos básicos da linguagem que são mais que suficientes para construir os métodos heurísticos, é claro que nada impede o leitor de buscar conhecimento que vai além deste compêndio, inclusive somos favoráveis a isso, pois o título deixa claro que trata-se de uma "introdução"aos assuntos aqui abordados. Na parte II são abordados os conceitos relacionados a aplicação do que foi aprendido na parte I, na construção dos métodos heurísticos, aplicado aos problemas de otimização.

Além disso, como o objetivo da parte I é tratar do básico da linguagem, palavras como: otimização, heurística, entre outras palavras relacionadas ao assunto da parte II, nem mesmo são mencionadas na parte I, inclusive os exercícios não são relacionadas à otimização, por um motivo simples, para iniciar na programação de heurísticas, é preciso ter os conceitos, em sua completude, da parte I, independente da origem do conhecimento. De forma que, programadores já com alguma experiência em linguagem Python, podem avançar direto para a parte II.

Desejamos uma boa leitura.

AULA 1

Introdução à Linguagem Python

Metas da Aula

1. Discutir um pouco sobre os programas em linguagem Python, codificação e compilação.
2. Entender e praticar os conceitos da sintaxe utilizada na linguagem de programação Python.
3. Entender e praticar os conceitos básicos relacionados à linguagem, como: definição e uso de variáveis, operadores aritméticos e operações de entrada e saída.

Ao término desta aula, você será capaz de:

1. Escrever programas em linguagem Python que sejam capazes de resolver problemas simples que envolvam, por exemplo, um cálculo matemático ou a entrada e saída de informações.
2. Escrever programas que manipulem informações em variáveis.

1.1 Programação em Linguagem Python

Para iniciar no mundo da Linguagem Python é importante ter um entendimento mínimo de como funciona o processo de construção de um programa. Em geral, quando se trata de linguagens de alto nível, que é o caso do Python, há a necessidade de um programa intermediário para traduzi-lo em linguagem de baixo nível. Há dois tipos: **interpretadores** e **compiladores**. O Python é considerado uma linguagem interpretada (BORGES, 2010), que funciona conforme apresentado na figura 1.

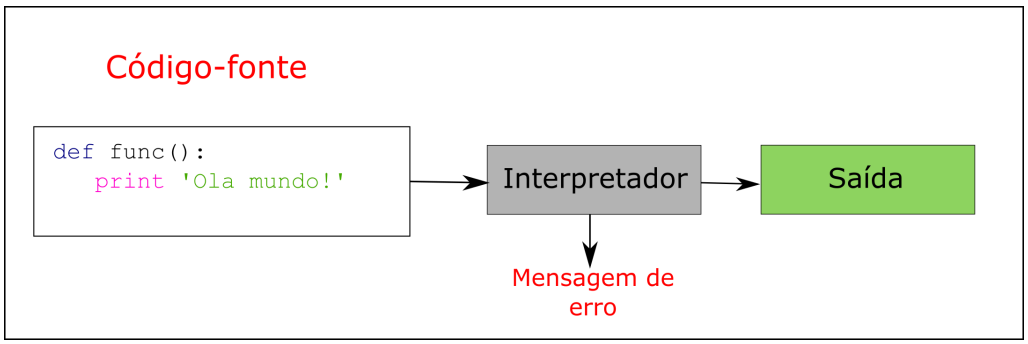

Figura 1 – Etapas da construção de uma aplicação

1.1.1 Etapa 1: Escrita do código-fonte

É nesta etapa que o programador realiza o esforço de escrever o código que dará origem ao programa final, conforme pode ser visto na Figura 1. Este código deve seguir regras rígidas para que o compilador tenha sucesso ao construir o programa. Este conjunto de regras ou formato que deve ser seguido é denominado como: sintaxe da linguagem. O objetivo deste livro é orientar o leitor neste momento em que o código-fonte é escrito, ou seja, como escrever seguindo a sintaxe correta da linguagem Python.

1.1.2 Etapa 2: Interpretação do programa

Após escrever o código-fonte, o programador deve executá-lo por meio de um interpretador. Existem vários tipos de interpretadores, ao instalar o Python, será disponibilizado e configurado um interpretador que possibilitará a sua execução. Outro detalhe importante é que, por ser uma linguagem interpretada, o Python lê e e executa as linhas do código-fonte, uma a uma. Ao acionar a execução do programa podem ocorrer duas situações: Na primeira situação, o interpretador não encontra erros no código-fonte e avança para etapa seguinte. Na segunda situação o interpretador encontra erros, neste caso, a operação é abortada e são exibidas mensagens com os erros que foram encontrados.

O programador deve então analisar as mensagens para corrigir os erros no código-fonte e refazer o processo de executar o programa. O interpretador pode identificar também código-fonte que, apesar de não possuir erros na sintaxe, mas que levantam suspeita de que, ao executar o programa, o mesmo irá se comportar de forma inesperada em determinadas situações. Neste caso, o interpretador avança para a próxima etapa, contudo, ele exibe avisos dos possíveis problemas que podem ocorrer.

Para executar um programa em linguagem Python, você pode utilizar uma IDE[1] de desenvolvimento, como: **IDLE**, **Spyder**, **Komodo-Edit**, dentre outras. Em geral, as IDE's disponibilizam o comando para acionar o interpretador de forma simples. Porém, o interpretador pode também ser acionado por comando em modo texto.

1.1.3 Etapa 3: Execução do Programa

Se as etapas 1 e 2 ocorrerem sem erros, então ao final deste processo tem-se a execução do programa. Para testar então a aplicação criada, basta interagir com o programa em execução.

1.2 Uso interativo x Execução por Scripts

Visto que o Python é uma linguagem interpretada, pode-se executar os programas de duas formas, por meio de scripts que possuam blocos de código-fonte, funções entre outros ou pode-se também executar de forma interativa com o uso de um **Shell** para execução de instruções do Python. O uso com script, segue o formato que no geral é utilizado em todas as linguagens de programação, ou seja, escreve-se o código-fonte desejado em um arquivo e, por meio de uma IDE, executa-se este código em um interpretador (MCKINNEY, 2013, p. 60).

No uso interativo, a diferença reside no fato de que não é necessário ter o código-fonte pronto, nem é necessário ter um arquivo salvo com o código-fonte. O programador pode escrever o código e executar a cada linha escrita, pode inclusive ir interagindo com o resultado obtido e até mesmo corrigindo o código que não funcionar como esperado. Inclusive, os valores armazenados em variáveis serão preservados durante o ciclo de vida do Shell, ou seja, valores que tenham sido armazenados em variáveis pelo programador durante o uso do modo interativo, serão preservados enquanto o Shell estiver em execução, após finalizar (fechar) a aplicação do Shell, os valores armazenados serão todos perdidos.

1.3 Meu primeiro Programa em Python

Agora que explicou-se como funciona o ciclo de construção de um aplicativo, pode-se dar os primeiros passos e construir o primeiro programa em Python. Como mencionado antes, para que o interpretador tenha sucesso em executar o programa, é preciso escrever uma sequência de código organizado e coeso de tal forma que seja capaz de resolver um problema logicamente. Outra questão importante que deve ser levada em consideração, é que um programa pode ser desenvolvido em módulos distintos e/ou subprogramas, neste caso, é necessário indicar qual o ponto de partida do programa, ou seja, em que ponto do código-fonte o programa deve iniciar suas instruções, mas isso é assunto para a aula 5. De forma que, nesta aula os códigos serão mais simples e diretos, podendo-se utilizar um Shell para a execução. Para iniciar, observe o código a seguir:

```
1 print 'Ola mundo'
```

Este primeiro código tem uma função simples, imprimir uma mensagem na tela, no caso, a mensagem **Ola mundo**. Para testar esse programa, selecione a IDE[2] de sua

[1] *Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado
[2] Não faz parte do escopo deste livro a abordagem do uso da IDE

preferência, digite o código e execute. No geral, as IDE's disponibilizam uma área para visualizar a saída do programa.

O código-fonte apresentado, pode ser dividido em duas partes, a primeira é o comando ou instrução, no caso, **print**, a segunda parte, a mensagem, faz parte dos argumentos ou parâmetros que devem ser informados ao comando. O comando **print** precisa receber os argumentos para que ele "saiba", o que deverá imprimir. Ele é capaz de receber variados tipos de argumentos. Para utilizar a função **print** corretamente, é preciso seguir a sintaxe a seguir:

```
1 //Sintaxe:
2 print argumentos
```

A função **print** do Python é capaz de receber o argumento de vários tipos e de várias formas. Veja a seguir alguns exemplos de como pode-se utilizar a função **print**:

```
1  >>> print 'Ola mundo'
2  Ola mundo
3  >>> print "Ola mundo"
4  Ola mundo
5  >>> print 1
6  1
7  >>> print 1 + 1
8  2
9  >>> print 'Ola' 'mundo'
10 Olamundo
11 >>> print 'Ola' + 'mundo'
12 Olamundo
13 >>> print 'Ola' + ' mundo'
14 Ola mundo
```

As várias instruções apresentadas no código-fonte foram executadas em modo interativo, isso pode ser observado, visto que, ao escrever cada instrução o resultado é apresentado na sequência, observe, por exemplo, a linha 1 e a linha 2, em que, a primeira traz a instrução e a segunda o resultado da execução da instrução. Veja também que a função **print** pode receber argumentos em formato de texto ou número, como nas linhas 1 e 5. Outra variação é que, ao informar um texto para a função print, pode-se utilizar aspas simples ou aspas duplas, como apresentado nas linhas 1 e 3.

Também é possível executar operações matemáticas com os argumentos, como é o caso da linha 7. Observe também que, pode se concatenar texto, somando os argumentos, linha 11, ou apenas informando mais de um argumento de texto, linha 9. Cabe lembrar que, ao concatenar texto, é preciso ter o cuidado, quando necessário, em garantir que o texto terá a devida separação com espaços, como exemplificado na linha 13.

1.4 Valores e tipos

Uma das coisas mais importantes em programação é a manipulação de valores, pois, em geral, desenvolvemos programas para resolver problemas do mundo real, que, por sua vez, é repleto de objetos que possuem valores e atributos, exemplo: o peso é um atributo possível para uma infinidade de objetos, que por sua vez pode assumir uma infinidade de valores conforme o objeto.

Da mesma forma, cada atributo, pode assumir valores, que em geral, tem um mesmo tipo, exemplo: o atributo idade só pode assumir valores numéricos, que geralmente,

serão inteiros positivos, pois não faria sentido uma pessoa, por exemplo, ter uma idade negativa e geralmente ao nos referirmos à idade, utilizamos apenas números inteiros. Quando falamos do atributo nome, neste caso, é comum que os valores sejam apenas no formato texto, como: "Ana", "Carlos", "banana", "carro", etc., dificilmente alguém nomearia algo ou alguém como 1, não é mesmo? Mesmo que haja explicitamente o desejo em denominar algo com o valor numeral, é comum que seja utilizada a forma em extenso para se referir ao numeral, ou seja, "Um", ao invés de 1.

Em Python, podemos conhecer o tipo de um valor por meio da instrução: **type()** (DOWNEY; ELKNER; MEYERS, 2010, p. 17), veja a seguir a sintaxe para uso da função **type**:

```
1 //Sintaxe:
2 type(argumento)
```

Basta então informar o comando **type** e entre parênteses informar o argumento. Veja a seguir alguns exemplos de uso.

```
1 >>> type('Ola mundo')
2 str
3 >>> type(1)
4 int
5 type(1.5)
6 float
7 type(True)
8 bool
```

Observe as linhas 1 e 2 no exemplo apresentado, note que, na linha 1 foi solicitado o tipo de um texto e o resultado foi apresentado na linha 2, no caso **str**, isso porque 'str' é a abreviação de 'string' que é a denominação dada à cadeia de caracteres, no qual os textos são formados. Na linha 3 e 4, foi solicitado o tipo para o valor **1**, como este valor é um inteiro, então o resultado é apresentado na linha 4, como **int**.

Da mesma forma é apresentado o tipo para o valor 1.5, como sendo **float**, que equivale ao real, do conjunto dos reais, note que o separador de decimal é o ponto. Na linha 7 foi solicitado o tipo para **True**, mas o que é **True**? Equivale à "Verdadeiro", e o seu antônimo é **False**, que equivale à "Falso". E o tipo no Python é o **bool**, que é a abreviação para booleano, tipo este utilizado para valores lógicos.

1.5 Variáveis

Para entender claramente o papel das variáveis, basta associar aos recipientes do mundo real. Se você deseja guardar água, então provavelmente você irá utilizar uma jarra, se você deseja guardar biscoitos, então provavelmente você irá utilizar um pote hermético, se quiser guardar ovos, então irá utilizar uma caixa de ovos. Da mesma forma, utiliza-se as variáveis em um programa de computador, para guardar. A diferença é que no caso do programa, são armazenados dados, mas, assim como no mundo real, os objetos exigem recipientes diferentes e adaptados, os dados também o exigem.

Para exemplificar, se é necessário armazenar um dado que é pertencente ao conjunto dos números reais, como o valor ou o peso de um produto, então é necessário uma variável que seja compatível com este dado, ou seja, que admita valores reais, desta forma, assim como em outras linguagens, o Python requer que a variável seja compatível com a informação armazenada, contudo, não é necessário definir o tipo de dado ao declarar uma variável, pois o Python se encarrega de determinar o tipo no

momento em que a primeira informação é armazenada na variável. Este processo é conhecido como tipagem dinâmica (COELHO, 2007, p. 13). Para definir o tipo, basta fazer uma atribuição de valor no momento da declaração, assim, declaramos a variável apenas quando vamos utilizá-la. No Python, utiliza-se o sinal de igual (=) para efetuar atribuições. Veja a seguir alguns exemplos de definição de variáveis.

```
1 >>> idade = 37
2 >>> nome = 'Carlos'
3 >>> valor = 13.5
```

Na linha 1 do exemplo, foi definida uma variável, cujo nome é *idade*, como o valor atribuído é um número inteiro, então, automaticamente, o Python determina o seu tipo como **int**. Na linha 2 foi declarada a variável, cujo nome é *nome*, e o valor atribuído é "Carlos". Por fim, na linha 3 foi declarada a variável *valor* e atribuído o número **13.5**, que é um número do conjunto dos reais. Podemos confirmar o tipo dessas variáveis pelo comando **type()** já aprendido. Veja a seguir:

```
1 >>> type(numero)
2 int
3 >>> type(nome)
4 str
5 >>> type(valor)
6 float
```

Veja que o comando **type()** pode receber como argumento uma variável também, e neste caso, informar qual o seu tipo. Em Python uma variável pode inclusive mudar de tipo dinamicamente conforme os valores que lhe são atribuídos. Veja um exemplo a seguir:

```
1 >>> numero = 1
2 >>> type(numero)
3 int
4 >>> numero = 1.5
5 >>> type(numero)
6 float
```

Observe que na linha 1 foi atribuído um valor inteiro à variável *numero*, assim, ao acionar o comando **type()** na linha 2, foi retornado o tipo **int** na linha 3, como era de se esperar, contudo, ao atribuir um valor real à mesma variável *numero* na linha 4 e invocar o comando **type()**, note que o tipo da variável mudou, pois o resultado retornado foi **float** na linha 6, assim, a mudança ocorreu dinamicamente, apenas atribuindo um valor com tipo diferente, no caso real, à variável.

Quando é preciso definir mais de uma variável na mesma instrução, pode-se fazer isso apenas separando o nome das variáveis por virgula e as suas respectivas atribuições, ou quando o valor atribuído é igual, então basta utilizar o sinal de igual igualando todas as variáveis (COELHO, 2007, p. 9), como os exemplos a seguir:

```
1 >>> numero1, numero2, numero3 = 1, 2, 3
2 >>> numero1
3 1
4 >>> numero2
5 2
6 >>> numero3
7 3
8 >>> numero1 = numero2 = numero3 = 0
```

```
9  >>> numero1
10 0
11 >>> numero2
12 0
13 >>> numero3
14 0
```

Veja que na linha 1 foram declaradas 3 variáveis, *numero1*, *numero2* e *numero3* e os valores atribuídos 1, 2 e 3 respectivamente. Como pode ver, os nomes das variáveis foram separados por virgula e da mesma forma, após o sinal de igual, responsável pela atribuição, os valores atribuídos também foram separados por virgula. As linhas que seguem, entre 2 e 7, são responsáveis apenas pela verificação dos valores atribuídos.

Mas, e se o valor a ser atribuído às diferentes variáveis é o mesmo? Neste caso, pode-se efetuar a declaração conforme o exemplo da linha 8, note que os nomes das variáveis são separados pelo sinal de igual, ou seja, neste caso, o objetivo não é separar a declaração das variáveis, mas atribuir uma variável a outra e no final da instrução o valor será atribuído. Na prática a instrução irá executar primeiro a atribuição do valor à última variável e na sequência a penúltima variável recebe o valor atribuído à última e assim por diante até que a primeira variável também receba o valor.

Assim como em outras linguagens, o Python possui várias regras para a definição do nome das variáveis (COELHO, 2007, p. 11), para esclarecer o que não é permitido ao definir uma variável, veja os exemplos apresentados na tabela 1, em que a coluna da esquerda é a forma não permitida em Python, a coluna do meio é o motivo pelo qual não é permitido e a coluna da direita, traz uma sugestão para fazer a declaração sem erro. Além disso, a tabela 2 descreve a lista de palavras reservadas que não podem ser utilizadas na nomenclatura das variáveis.

Tabela 1 – Situações incorretas na nomenclatura das variáveis

Forma incorreta	*Motivo*	*Sugestão de correção*
4nota = 10	Não é permitido iniciar o nome da variável com números.	nota4 = 10
and = 0	Não é permitido utilizar palavra reservada como nome de uma variável.	vAnd = 0
vinte% = 20	Não é permitido utilizar caracteres especiais como %, @, #, $, &, etc.	vintePercent = 20
idade pes = 27	Não é permitido separar os nomes compostos em variáveis.	idade_pes = 27

Fonte: Os autores

Tabela 2 – Palavras reservadas da linguagem Python

Palavras Resevadas					
and	**as**	**assert**	**break**	**class**	**continue**
def	**del**	**elif**	**else**	**except**	**exec**
finally	**for**	**from**	**global**	**if**	**import**
in	**is**	**lambda**	**not**	**or**	**pass**
print	**raise**	**return**	**try**	**while**	**yield**

Fonte: Adaptado de Coelho (2007, p. 4)

Outro ponto que deve ser levado em consideração ao definir as variáveis, é que a linguagem Python é *"case sensitive"*, ou seja, letras maiúsculas e minúsculas correspondentes são consideradas diferentes, desta forma, ao utilizar a variável, esta, deverá ser escrita exatamente como foi declarada, pois caso contrário o interpretador irá entender que se trata de outra variável (BIRD; KLEIN; LOPER, 2009, p. 15).

1.6 Operadores do Python

Como já visto, para armazenar dados em variáveis, é preciso fazer atribuição. Para isso, deve-se utilizar o operador de atribuição, que na linguagem Python é o sinal de igualdade, **"="**. Além de atribuir valores simples, como já exemplificado, pode-se atribuir às variáveis, expressões matemáticas. Para isso, é preciso utilizar os operadores aritméticos relacionados na tabela 3.

Tabela 3 – Operadores aritméticos e unários

Operador	*Operação Matemática*
+	Adição
-	Subtração.
*****	Multiplicação
/	Divisão
//	Divisão (obtém a parte inteira da divisão)
%	Módulo (obtém o resto da divisão)
*****	Potência

Fonte: Adaptado de Lutz e Ascher (2007, p. 82)

Ao escrever uma expressão matemática em Python deve ser considerada a ordem das operações em matemática, exemplo, na inexistência de parênteses, as operações de multiplicação e divisão serão realizadas antes das operações de adição e subtração. Quando houver a necessidade de alterar a precedência das operações, então utilize os parênteses. Veja a seguir alguns exemplos de uso dos operadores aritméticos:

```
1  >>> contador = 2
2  >>> valor1 = 300
3  >>> valor2 = 400
4  >>> totalSom = valor1 + valor2
5  >>> totalSom
6  700
7  >>> totalMult = valor1 * valor2
8  >>> totalMult
9  120000
10 >>> resul = (totalMult + totalSom) * contador
11 >>> resul
12 241400
13 >>> potValor = valor1 ** contador
14 >>> potValor
15 90000
```

Nas primeiras três linhas, foi feita a declaração de variáveis e atribuição simples de valor. Na linha 4 a variável *totalSom* recebe a soma das variáveis *valor1* e *valor2*. E na linha 7 a variável *totalMult* recebe a multiplicação das variáveis *valor1* e *valor2*. Na

linha 10 a variável *resul* recebe a operação que envolve soma e multiplicação, sendo que a operação de soma é priorizada em relação à operação de multiplicação.

É comum em programação a necessidade de acumular o valor já armazenado em uma variável ao fazer o cálculo. Neste caso, a variável recebe a própria variável acompanhada da operação de cálculo que pode ser uma soma, uma subtração, etc. Veja o exemplo de um acumulo com soma.

```
1 >>> totalSom = 200
2 >>> valor1 = 300
3 >>> valor2 = 400
4 >>> totalSom = totalSom + (valor1 - valor2)
5 >>> totalSom
6 100
```

Note que para acumular a atribuição com o valor que já estava armazenado na variável *totalSom*, a atribuição é feita de *totalSom* adicionado da expressão matemática à *totalSom*. Assim, o valor que já estava armazenado em *totalSom* é acumulado ao invés de ser substituído. Esta operação funciona na linguagem Python, contudo há uma forma mais elegante para fazê-la. Veja a seguir a mesma operação com o ajuste necessário.

```
1 >>> totalSom = 200
2 >>> valor1 = 300
3 >>> valor2 = 400
4 >>> totalSom += valor1 - valor2
5 >>> totalSom
6 100
```

Ao analisar a linha 4 observa-se que a variável *totalSom* no lado direito da igualdade foi suprimida, pois o operador de soma antes da igualdade já indica que deseja-se que ela seja acumulada. Além disso, ao remover a variável do lado direito da igualdade, os parênteses também se tornaram desnecessários. A tabela 4 apresenta os operadores de atribuições disponíveis.

Tabela 4 – Operadores de atribuição

Operador	*Operação Matemática*
=	Atribuição simples
+=	Atribuição acumulando por soma.
-=	Atribuição acumulando por subtração
***=**	Atribuição acumulando por multiplicação
/=	Atribuição acumulando por divisão
%=	Atribuição acumulando por módulo
****=**	Atribuição acumulando por potência

Fonte: Adaptado de Coelho (2007, p. 18)

1.7 Entrada e Saída

Para um programa de computador é fundamental a interação com dispositivos de entrada e de saída, pois é por meio da entrada que os programas recebem os dados e por meio da saída que são disponibilizadas às informações aos usuários. Isso é um princípio

básico da computação. Nas primeiras aulas deste livro, será utilizada a interação com o teclado como dispositivo de entrada e o monitor como dispositivo de saída. A partir da aula 6, será utilizada também a entrada de dados por meio de arquivos.

1.7.1 Função print

Um pouco já foi falado dessa função no primeiro programa. Mas, agora é o momento oportuno para tratar melhor sobre ela. A função **print** permite realizar a impressão de textos no monitor, ou seja, é responsável pela saída de informações. Esta função possui um número variado de parâmetros, tantos quantos forem necessários. Veja a seguir a sintaxe para utilizar corretamente a função **print**.

```
1 #Sintaxe:
2 print argumentos
```

A função **print** é capaz de receber vários argumentos, mas obrigatoriamente, ao menos 1 deve ser informado, assim, no mínimo deve-se informar um texto para ser impresso. Os próximos argumentos são opcionais, pois nem sempre é necessário apresentar uma informação em conjunto do texto. Um exemplo de como utilizar a função **print** apenas com o primeiro argumento foi apresentado no primeiro programa com a mensagem 'Ola mundo'.

Há duas formas de utilizar a função **print** com mais de um argumento. A primeira forma é separando os argumentos por virgula, assim, se você deseja, por exemplo imprimir a combinação de um texto com um número armazenado em uma variável, basta informar o argumento texto, separar por virgula e informar o segundo argumento com a variável. Para mais argumentos, basta separar por virgula. Veja a seguir alguns exemplos:

```
1 >>> x, y, z = 1, 2, 3
2 >>> print "x =", x, "y =", y, "z =", z
3 x = 1 y = 2 z = 3
4 >>> print "x =", x, "\ny =", y, "\nz =", z
5 x = 1
6 y = 2
7 z = 3
```

No exemplo, observa-se que na linha 1 foram declaradas três variáveis, *x*, *y* e *z* com a atribuição dos valores **1**, **2** e **3** respectivamente. Na linha 2 foi escrita a instrução para a impressão das 3 variáveis, sendo que a função **print** recebeu os argumentos separados por virgula, como mencionado. Note que, primeiro foi informado um texto entre aspas duplas para indicar que será impresso o valor da variável *x*, na sequência, é informada a variável *x* como argumento, ou seja, após a impressão do primeiro texto será impresso o valor da variável *x*. Os próximos argumentos seguem o mesmo padrão para as variáveis *y* e *z*. Observe também que foram informados 6 argumentos ao todo nesta instrução.

O resultado da execução da instrução na linha 2 pode ser visto na linha 3. Veja que os valores armazenados nas variáveis foram combinados com o texto informado nos argumentos da função **print**. Na linha 4 a função **print** foi utilizada da mesma forma, a única diferença é que o formatador de quebra de linha, **"\n"**, foi combinado com o texto entre aspas duplas. Mas, qual foi o objetivo? Pode ser visualizado nas linhas seguintes, 5, 6 e 7, em que o resultado impresso é praticamente igual, exceto pelo fato de que os valores impressos para as variáveis foram separados por quebra de linha. Então, quando precisar imprimir com quebra de linha, basta utilizar o **"\n"**.

A segunda forma de utilizar a função **print** é similar ao uso do comando **printf** na linguagem C, em que são utilizados os formatadores para indicar a posição em que será impresso o conteúdo da variável que será combinada com o texto. Veja a seguir alguns exemplos:

```
1 >>> x, y, z = 1, 2, 3
2 >>> print "x = %d y = %d z = %d" % (x, y, z)
3 x = 1 y = 2 z = 3
4 >>> print "x = %d\ny = %d\nz = %d" % (x, y, z)
5 x = 1
6 y = 2
7 z = 3
```

Primeiro, observe a linha 2, foi informado o texto entre aspas como primeiro argumento, note que, neste argumento, foi colocado todo o texto que deve ser impresso, mas no lugar do valor das variáveis, que não é fixo, foi indicado o formatador **"%d"** para as três variáveis. Este formatador será substituído pelo valor contido em cada variável que foi informada no segundo argumento. Agora observe na linha 3 que o resultado impresso é exatamente igual ao que foi impresso no primeiro exemplo, ou seja, o que mudou foi apenas a forma de uso da função **print**. Nas linhas seguintes, observa-se apenas a presença adicional do formatador de quebra de linha combinado com o texto.

Neste exemplo foi utilizado o **%d** porque os valores armazenados nas variáveis são inteiros, mas, para cada tipo de valor há um formatador diferente, conforme apresentado na tabela 5.

Tabela 5 – Formatadores de tipo em Python

Formatador	*Tipo do valor*
%s	string
%d	inteiro
%o	octal
%x	hexadecimal
%f	real
%e	real exponencial
%%	sinal de percentagem

Fonte: Adaptado de Borges (2010, p. 35)

1.7.2 Função raw_input()

Como visto a função **print** é responsável pela saída das informações do programa. Como fazer então para entrar com dados na fronteira do programa? Neste caso, deve-se utilizar a função **raw_input()**. Esta função permite que os dados digitados pelo usuário do programa sejam armazenados nas variáveis do programa e o seu uso é bem simples (CRUZ, 2017, p. 29). Veja a seguir a sintaxe de seu uso e na sequência um exemplo.

```
1 #Sintaxe:
2 variavel = raw_input("texto com orientacao para o usuario")
```

```
1 >>> s = raw_input("Informe algum texto:")
2 Informe algum texto:texto digitado de exemplo
3 >>> print s
```

```
4 texto digitado de exemplo
```

Note que, conforme a sintaxe, basta declarar uma variável e pela atribuição, invocar a função **raw_input()** com um texto como argumento, o objetivo deste texto é orientar o usuário em relação ao que ele deverá preencher. Na linha 1 do exemplo, é feito exatamente isso, foi declarada a variável *s* e pela atribuição, foi invocada a função **raw_input()** com o argumento **"Informe algum texto:"**. O resultado da execução pelo modo interativo pode ser visto na linha 2, note que primeiro aparece o texto informado como argumento da função **raw_input()**, depois o texto digitado pelo usuário. Para confirmar que o texto foi armazenado na variável, a linha 3 do exemplo traz a impressão da variável *s* e o resultado está na linha 4. Veja agora outro exemplo:

```
1 >>> numero = raw_input("Informe um numero:")
2 Informe um numero:50
3 >>> type(numero)
4 str
```

Observe que na linha 1 do exemplo, foi realizada a declaração da variável *numero* e a invocação da função **raw_input()**. Na linha 2, o resultado da execução, note que no exemplo foi digitado o valor 50, contudo, observe que na linha 3, ao obter o tipo da variável *numero* foi apresentado **str**. Isso ocorreu porque os valores vindos da função **raw_input()** são do tipo **string**. Como fazer então? É preciso converter os valores, para isso podemos utilizar as funções **int()** para converter para valores inteiros e **float()** para converter para valores reais (CRUZ, 2017, p. 29). Veja a seguir o exemplo:

```
1 >>> numero = int(raw_input("Informe um numero:"))
2 Informe um numero:50
3 >>> type(numero)
4 int
```

Conforme pode-se observar na linha 1, ao invocar a função **raw_print()**, teve-se o cuidado de converter o seu resultado com a função **int()** e somente após convertido, é realizada a atribuição na variável *numero*. Neste caso, o resultado ocorre conforme esperado, pois note que a variável possui conteúdo do tipo inteiro agora, conforme a linha 4. Como, até este momento, já foi explicado em como declarar variáveis, fazer operações de atribuição e aritméticas, e utilizar funções de entrada e saída, é possível fazer o primeiro exercício.

1.7.2.1 Exercício de Exemplo

Faça um programa em Python que receba dois números inteiros e ao final imprima a soma deles.

```
1 numero1 = int(raw_input("Informe o primeiro numero inteiro:"))
2 numero2 = int(raw_input("Informe o segundo numero inteiro:"))
3
4 soma = numero1 + numero2
5
6 print "Resultado da soma: ", soma
```

As linhas 1 e 2 do programa apresentado no exercício de exemplo, são responsáveis pela declaração das variáveis, conforme pedido no enunciado do exercício, sendo *numero1* e *numero2* para receber os dois valores inteiros, a atribuição dos valores com o uso da função **raw_input()** e a conversão para valores inteiros com a função **int()**. Na

linha 4 foi declarada a variável *soma* e realizada a atribuição da operação de soma dos valores contidos nas variáveis *numero1* e *numero2*. Por fim, a impressão do resultado da soma é realizada na linha 6. Desta forma, o programa contempla a entrada dos dados (linhas 1 e 2), o processamento (linha 4) e saída dos dados (linha 6) e atende aos requisitos do enunciado do exercício de ler os dois valores inteiros e imprimir a soma deles.

1.7.3 Comentários

Até este ponto do livro foi feito pouco uso dos comentários, pois a parte final da primeira aula foi reservada para falar sobre eles. Mas para que servem os comentários? Bem, até o momento, ficou claro que no contexto da programação de computadores, os compiladores e interpretadores são muito criteriosos e são exigentes na escrita do programa, não deixam nenhum erro sintático passar despercebido. Mas, em programas com 10 mil, 20 mil ou mais linhas de códigos, é importante, principalmente em trechos menos intuitivos, comentar (explicar) sobre o que foi escrito, mas como o compilador não aceita algo diferente da sintaxe da linguagem de programação misturada ao código, em geral, as linguagens de programação disponibilizam "indicadores"para de certa forma, dizer ao interpretador: "despreze esse trecho!", assim, ao indicar qual trecho do código que o compilador deve desprezar, pode-se fazer uso deste para escrever em qualquer formato, incluindo a linguagem formal ou informal, que o interpretador não irá gerar erros relacionados àquele trecho. Este recurso é chamado de "comentário".

Veja então quais são os "indicadores"que a linguagem Python disponibiliza para fazer comentários no código. A seguir um exemplo.

```
1  #trecho responsavel pela entrada dos dados
2  numero1 = int(raw_input("Informe o primeiro numero inteiro:"))
3  numero2 = int(raw_input("Informe o segundo numero inteiro:"))
4  '''
5   O trecho a seguir e responsavel pela soma dos valores de num1
6   e num2 informados pelo usuario do programa
7  '''
8  soma = numero1 + numero2
9
10 print "Resultado da soma: ", soma  #impressao do resultado
```

Foram apresentadas duas formas de utilizar comentários em linguagem Python, a primeira forma é apresentada na linha 1 e 10 do exemplo, trata-se do uso do símbolo sustenido, também conhecido como tralha, **#**, este formato é o mais comum e permite comentar o código a partir do ponto em que o sustenido é incluído e apenas na linha em que ele foi adicionado (BORGES, 2010, p. 15). Uma vantagem do uso do sustenido é que não é necessário indicar aonde termina o comentário, uma vez que, pela sua natureza apenas uma linha ou parte dela é comentada.

Contudo se for preciso escrever um comentário com mais de uma linha, então o segundo formato é o mais indicado, pois, neste caso, basta colocar o indicador de onde o comentário deve iniciar e depois colocar o indicador de onde o comentário termina, assim, é possível escrever comentários com várias linhas apenas indicando o ponto de início e término. A linha 5 apresenta o exemplo deste formato de comentário, note que neste caso, o indicador de início é ''' e o indicador de término é ''', conforme a linha 7.

A partir deste ponto do livro, este recurso será frequentemente utilizado, para explicar exercícios resolvidos que forem apresentados. Pois, em alguns casos, é muito útil quando a explicação está bem ao lado do código escrito. Contudo, assim como no cotidiano de um programador, os comentários serão inseridos no código com moderação.

1.8 Resumo da Aula

Nesta primeira aula foram apresentados os principais conceitos introdutórios da linguagem Python, que serão necessários para dar continuidade no estudo. Para desenvolver um programa básico em Python, minimamente é preciso saber:

- Declarar variáveis
- Atribuir valores a variáveis
- Efetuar operações aritméticas
- Realizar entrada de dados
- Realizar saída de informações

Assim, este foi o objetivo desta aula, aprender objetivamente como realizar estas operações para que, a partir da próxima aula, seja possível aprender os conceitos mais avançados da linguagem.

No que diz respeito à declarar variáveis, é importante ficar atento ao fato de que a linguagem Python é *case sensitive*, ou seja, uma variável iniciando com letra maiúscula é diferente de uma variável com mesmo nome, mas iniciando com letra minúscula, ao ficar atento a isso, em geral, você conseguirá diminuir uma boa parte dos problemas que poderá enfrentar ao se deparar com mensagens de erro apresentadas pelo interpretador.

1.9 Exercícios da Aula

Os exercícios desta lista foram Adaptados de Lopes e Garcia (2002, p. 38-52).

1. Faça um programa em Python que imprima o seu nome.
2. Faça um programa em Python que imprima o produto dos valores 30 e 27.
3. Faça um programa em Python que imprima a média aritmética entre os números 5, 8, 12.
4. Faça um programa em Python que leia e imprima um número inteiro.
5. Faça um programa em Python que leia dois números reais e os imprima.
6. Faça um programa em Python que leia um número inteiro e imprima o seu antecessor e o seu sucessor.
7. Faça um programa em Python que leia o nome o endereço e o telefone de um cliente e ao final, imprima esses dados.
8. Faça um programa em Python que leia dois números inteiros e imprima a subtração deles.
9. Faça um programa em Python que leia um número real e imprima ¼ deste número.
10. Faça um programa em Python que leia três números reais e calcule a média aritmética destes números. Ao final, o programa deve imprimir o resultado do cálculo.
11. Faça um programa em Python que leia dois números reais e calcule as quatro operações básicas entre estes dois números, adição, subtração, multiplicação e divisão. Ao final, o programa deve imprimir os resultados dos cálculos.
12. Faça um programa em Python que leia um número real e calcule o quadrado deste número. Ao final, o programa deve imprimir o resultado do cálculo.
13. Faça um programa em Python que leia o saldo de uma conta poupança e imprima o novo saldo, considerando um reajuste de 2%.
14. Faça um programa em Python que leia a base e a altura de um retângulo e imprima o perímetro (base + altura) e a área (base * altura).
15. Faça um programa em Python que leia o valor de um produto, o percentual do desconto desejado e imprima o valor do desconto e o valor do produto subtraindo o desconto.
16. Faça um programa em Python que calcule o reajuste do salário de um funcionário. Para isso, o programa deverá ler o salário atual do funcionário e ler o percentual de reajuste. Ao final imprimir o valor do novo salário.
17. Faça um programa em Python que calcule a conversão entre graus centígrados e Fahrenheit. Para isso, leia o valor em centígrados e calcule com base na fórmula a seguir. Após calcular o programa deve imprimir o resultado da conversão.

$$F = \frac{9 \times C + 160}{5} \tag{1.1}$$

Em que:

- F = Graus em Fahrenheit
- C = Graus centígrados

18. Faça um programa em Python que calcule a quantidade de litros de combustível consumidos em uma viagem, sabendo-se que o carro tem autonomia de 12 km por litro de combustível. O programa deverá ler o tempo decorrido na viagem e a velocidade média e aplicar as fórmulas:

$$D = T \times V \tag{1.2}$$

$$L = \frac{D}{12} \tag{1.3}$$

Em que:

- D = Distância percorrida em horas
- T = Tempo decorrido
- V = Velocidade média
- L = Litros de combustível consumidos

Ao final, o programa deverá imprimir a distância percorrida e a quantidade de litros consumidos na viagem.

19. Faça um programa em Python que calcule o valor de uma prestação em atraso. Para isso, o programa deve ler o valor da prestação vencida, a taxa periódica de juros e o período de atraso. Ao final, o programa deve imprimir o valor da prestação atrasada, o período de atraso, os juros que serão cobrados pelo período de atraso, o valor da prestação acrescido dos juros. Considere juros simples.

20. Faça um programa em Python que efetue a apresentação do valor da conversão em real (R$) de um valor lido em dólar (US$). Para isso, será necessário também ler o valor da cotação do dólar.

AULA 2

Estruturas de Decisão

Metas da Aula

1. Entender e praticar os conceitos da sintaxe utilizada na linguagem Python para estruturas de decisão, operadores relacionais e lógicos.
2. Aplicar variadas situações relacionadas ao fluxo de decisão em programação objetivando cobrir diversificadas possibilidades vivenciadas na programação cotidiana.
3. Escrever programas que farão uso de fluxos de decisão no processamento dos dados.

Ao término desta aula, você será capaz de:

1. Escrever programas em linguagem Python que sejam capazes de resolver problemas que envolvam, por exemplo, a decisão entre um cálculo ou outro, dada uma determinada circunstância.
2. Escrever programas que manipulem as informações, considerando variados operadores relacionais e lógicos.

2.1 Estruturas de Decisão

Ao longo da aula 1, foram abordados os conceitos necessários para a criação de programas em Python com um único fluxo ou caminho. Ou seja, programas que irão sempre executar um determinado número de instruções sempre em um mesmo formato. Contudo, em geral, uma pequena parte dos problemas computacionais se limita a problemas com tal complexidade, grande parte das vezes os problemas dão origem a programas que possuem variados fluxos ou caminhos. Para que as instruções em um programa tomem um caminho diferente, é necessário que haja uma instrução responsável pela decisão de qual caminho tomar (COELHO, 2007; BORGES, 2010; CRUZ, 2017).

Na linguagem Python será abordada a instrução **if**, **elif** e **else**, mas antes é importante entender os operadores relacionais. Serão abordados também os conceitos sobre os operadores lógicos e sobre a indentação.

2.2 Operadores relacionais

Em linguagens de programação os operadores relacionais permitem estabelecer relação de comparação entre valores, exemplo: determinar se o valor 5 é menor que o valor 10. O uso dos operadores relacionais é imprescindível em estruturas de decisão e o Python possui 9 deles, conforme pode ser visto na tabela 6.

Tabela 6 – Operadores relacionais

Operador	*Nome*	*Exemplo*	*Significado do exemplo*
==	Igualdade	a == b	a é **igual** a b?
>	Maior que	a > b	a é **maior** que b?
>=	Maior ou igual que	a >= b	a é **maior ou igual** a b?
<	Menor que	a < b	a é **menor** que b?
<=	Menor ou igual que	a <= b	a é **menor ou igual** a b?
!=	Diferente de	a != b	a é **diferente** de b?
<>	Diferente de	a <> b	a é **diferente** de b?
is	É	a is b	a **é** b?
in	Está contido em	a in (a, b, c)	a **está contido em** (a, b, c)?

Fonte: Adaptado de (CRUZ, 2017, p. 30)

Como apresentado na tabela 6 há várias relações possíveis para se estabelecer em um programa Python. As 7 primeiras são as de uso mais comum, que permitem estabelecer as relações mais corriqueiras, como comparar se um valor é igual a outro ou menor que outro, ou ainda se um valor é diferente de outro. Observe que, para comparar se um valor é diferente, o Python disponibiliza dois operadores, **!=** e **<>**, mas que produzem o mesmo resultado. Veja alguns exemplos:

```
1 >>> 1 == 1
2 True   #retornou True para a comparacao de igualdade entre os valores
3 >>> 1 == 2
4 False
5 >>> 10 < 5
6 False #retornou False para a comparacao de menor que
7 >>> 'paulo' != 'ana'
8 True
9 >>> 'paulo' == 'ana'
```

```
10 False
11 >>> 10 <= 10
12 True
```

Na linha 1 do exemplo foi codificada a comparação do valor 1 com o valor 1, parece óbvio, é claro, mas o objetivo foi exemplificar o uso do operador relacional de igualdade. Quando o interpretador do Python retornou **True** na linha 2, é como se ele dissesse: "Essa comparação é verdadeira!". Observe agora que na linha 3 foi utilizado o mesmo operador para comparar o valor 1 com o valor 2, neste caso, o interpretador retornou **False** na linha 4, ou seja, ele está dizendo: "Essa comparação é falsa!".

Note agora, que na linha 7 foi utilizado o operador **!=** para comparar dois textos e o retorno na linha 8 é **True**, ou seja, verdadeiro. Mas, pera ai? Se os dois textos são diferentes não deveria retornar **False**? Não! Vamos pensar, é como se você estivesse perguntando ao interpretador: 'paulo' é diferente de 'ana'? Qual seria a resposta correta então? Verdade! 'paulo é diferente de 'ana', ou seja, a resposta para a sua pergunta é **True**. Desta forma, é importante ter o devido cuidado ao estabelecer essas relações lógicas. Nos exemplos apresentados foram comparados valores, mas e variáveis, podemos comparar? Veja a seguir:

```
1  >>> num1 = 10
2  >>> num2 = 20
3  >>> num1 == num2
4  False
5  >>> num1 != num2
6  True
7  >>> num1 < num2
8  True
9  >>> num1 > num2
10 False
```

Com este exemplo fica claro a diferença do uso do operador de atribuição, =, para o operador de comparação ==. O uso do == ao comparar é importante para o interpretador reconhecer quando você deseja atribuir e quando você deseja comparar. Veja que é possível comparar variáveis conforme as linhas 3, 5, 7 e 9, mas note que, o que é comparado com estes operadores não é a variável em si, mas o seu conteúdo, ou seja, o conteúdo atribuído às variáveis nas linhas 1 e 2 é o que determina a resposta de verdadeiro ou falso do interpretador.

Diferente dos primeiros 7 operadores, os dois últimos operadores **is** e **in**, apresentados na tabela 6, estão relacionados à comparação de objetos e não de valores, por exemplo, o operador **is** compara a identidade de dois objetos, ou seja, se eles tem a mesma identidade e não se tem o mesmo valor. O exemplo a seguir deixa isso bem claro.

```
1 >>> a = [1,2,3]
2 >>> b = [1,2,3]
3 >>> a == b
4 True
5 >>> a is b
6 False
```

No exemplo foram criados dois conjuntos, *a* e *b* com valores idênticos, assim, ao comparar os dois conjuntos, na linha 3, com o operador de igualdade, que compara o conteúdo, o resultado é **True**, na linha 4, pois, de fato, os conteúdos são iguais. Mas,

ao comparar, na linha 5, se *a* é *b*, o resultado é **False**, na linha 6, pois apesar de terem conteúdo igual, as variáveis não tem a mesma identidade. E o que dizer sobre o operador **in**? O operador **in** é utilizado para verificar se um conjunto possui um determinador valor. Veja o exemplo a seguir:

```
1  >>> a = [1,2,3]
2  >>> b = [1,2,3]
3  >>> 2 in a
4  True
5  >>> 5 in a
6  False
7  >>> a in b
8  False
9  >>> b = [[1, 2, 3], 2, 3]
10 >>> a in b
11 True
```

Novamente foram utilizados os dois conjuntos para exemplificar, conjunto *a* e *b*, inicialmente com valores iguais, linhas 1 e 2. Na linha 3, o operador **in** permitiu verificar se o valor 2 está contido no conjunto *a*, como isso é verdade, então a resposta é **True** na linha 4. Para validar um caso falso, a linha 5 verifica se o valor 5 está contido em *a* e a resposta esperada é apresentada na linha 6. Agora note a comparação na linha 7, ela retornou **False** na linha 8, porque? Porque apesar dos conteúdos de *a* e *b* serem iguais, não há em *b* um conteúdo igual ao de *a*. Como assim? Veja a linha 9, notou o ajuste feito em *b*? Agora verifique o resultado da comparação na linha 11, percebeu? Como o operador **in** compara se um conteúdo está contido, então a comparação ocorre com cada elemento do conjunto e não com todos os elementos ao mesmo tempo.

2.3 Cláusula **if**, **elif** e **else**

Agora que os operadores relacionais já são conhecidos, é possível avançar com as estruturas de decisão. A cláusula **if** permite estabelecer um controle de fluxo no programa de forma que o mesmo, possa escolher quando executar um determinado bloco de instruções ou não, ou ainda, optar por executar um bloco de instruções em vez de outro (COELHO, 2007, p. 28). Veja a seguir a sintaxe:

```
1 #Sintaxe:
2 if condicao:
3     instrucao
4 elif condicao:
5     instrucao
6 else:
7     instrucao
```

A sintaxe apresentada mostra como é a estrutura da cláusula **if**, note que, a primeira condição determinada pelo programador recebe a cláusula **if** como precedente, é possível, já que estamos falando de fluxos alternativos, que esta primeira condição não seja verdadeira, neste caso, o restante da estrutura será determinado pela quantidade de fluxos alternativos que se deseja, sendo que, para cada condição adicional que deve ser validada, o programador irá adicionar uma condição **elif** e ao final, quando todas as condições já tiverem sido validadas, restando apenas o caso contrário às demais, então o programador adicionará a cláusula **else**. Os exemplos serão apresentados desde a

forma mais simples de uso do **if** até as mais complexas. Veja a seguir um exemplo em forma de exercício para entender melhor.

2.3.1 Exercício de Exemplo

Faça um programa em Python que receba um número inteiro e verifique se este número é maior que 20, em caso afirmativo o programa deverá imprimir a mensagem: "Maior que 20".
Fonte: Adaptado de Lopes e Garcia (2002, p. 78)

```
1 #obtendo o numero inteiro
2 numero = int(raw_input("Informe um numero inteiro:"))
3
4 #validacao com a clausula if
5 if numero > 20:
6     print "Maior que 20"
```

Para resolver este exercício foi preciso declarar uma variável para armazenar um número inteiro, realizar a leitura com a função de entrada e após isso, verificar se o número é maior que 20, para imprimir ou não a mensagem. Ou seja, a decisão no programa está relacionada a executar ou não a instrução de impressão da mensagem dada uma determinada condição que foi estabelecida como: o número é maior que 20.

Até o momento em que o número é lido, o programa é similar ao que foi aprendido na aula 1. Nas linhas 5 e 6, foi incluída a cláusula **if** em sua forma mais simples de uso, em que, a condição é simples, apenas uma instrução é executada caso essa condição seja verdadeira, e além disso, não foi necessário o uso do **elif**, nem do **else**. A linha 5 traz o início da instrução representada pela palavra reservada **if**, logo após, está a condição **numero > 20**, sempre que houver uma condição, esta será acompanhada, como no exemplo, de um operador relacional, que no caso, é o sinal de maior, >. Este operador estabelece a relação entre os operandos. A tabela 6 apresenta os operadores relacionais disponíveis na linguagem Python, conforme já discutido nesta aula.

Assim, no exemplo anterior, quando o usuário do programa informar um número que seja maior que 20, a relação estabelecida na instrução da linha será verdadeira e, portanto, o comando da linha seguinte, que pertence ao bloco da cláusula **if**, será executado. No caso contrário, ou seja, no caso em que o usuário informa um número menor ou igual a 20, então a relação estabelecida entre os operandos será falsa e, portanto o programa irá saltar a linha 6, que pertence ao bloco da cláusula **if** e avançará para a linha posterior do programa, caso exista, não executará, assim, a instrução que imprime a mensagem na tela.

Como saber que a linha 6 pertence ao bloco de instruções da linha 5? Bem, este é um assunto que deve ser tratado agora, antes de dar sequência aos conceitos da cláusula **if**, pois ele é de suma importância no Python.

2.4 Indentação

A indentação no Python é obrigatória! Isso mesmo, se o programador, ao escrever o código não tomar os devidos cuidados com a indentação, o programa terá um mal funcionamento, pois é a indentação que determina quais instruções pertencem aos blocos e sub-blocos do programa escrito (CRUZ, 2017, p. 33). Assim, tomando o exemplo anterior de código, a linha 6 do código-fonte é um sub-bloco do código e isso foi determinado pelo espaçamento (recuo) que foi adicionado na instrução desta linha.

Desta forma, o interpretador, ao passar por este trecho de código ele saberá determinar que esta linha, no caso a 6, só será executada caso a condição da linha 5 seja verdadeira, pois foi adicionado à ela o espaçamento necessário para isso.

Este conceito é de suma importância em Python, pois pode facilmente produzir um mal funcionamento no programa. Observe a figura 2. Notou? O código-fonte nas duas caixas é exatamente igual, exceto pela indentação, e devido à essa diferença, o resultado produzido é diferente.

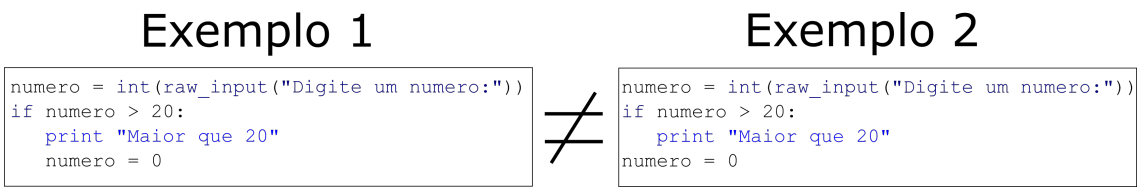

Figura 2 – Exemplos de diferença que a indentação pode produzir em Python

Observe que no exemplo 1 da figura 2 a instrução da quarta linha, só será executada caso a condição da linha 2 seja verdadeira, pois o recuo presente nesta instrução, indica que ela pertence ao bloco de instruções da cláusula **if**. Já no exemplo 2 da figura 2, não há o recuo na instrução da linha 4, ou seja, essa instrução pertence ao bloco principal de instruções e será executado mesmo que a condição da linha 2 não seja verdadeira. Assim, sempre que utilizar a indentação daqui em diante, lembre-se que está utilizando para expressar a relação de dependência que há entre um bloco de instruções com outro.

2.5 Voltando à Cláusula **if**, **elif** e **else**

Agora que os conceitos sobre a indentação em Python foram esclarecidos, é possível avançar no conhecimento das estruturas de decisão. No primeiro programa com a cláusula **if**, apresentado nesta aula, foi incluída apenas a validação de uma condição, mas, o que o programa fará se o resultado da expressão não for verdadeiro? Ou seja, o que ele fará se o valor informado pelo usuário foi menor ou igual a 20? Bem, no exemplo apresentado, o programa não fará nada! Pois, como não foi solicitado no enunciado do exercício anterior, não foi indicado o que o programa deveria fazer neste caso. A seguir, o exercício foi aprimorado para exemplificar o uso da cláusula **else**.

2.5.1 Exercício de Exemplo

Faça um programa em Python que receba um número inteiro e verifique se este número é maior que 20, em caso afirmativo o programa deverá multiplicar o valor por 2 e em caso negativo deve multiplicar por 4. Após realizar o cálculo o programa deve imprimir a mensagem: "Resultado: <valor do resultado>", em que <valor do resultado> deve ser substituído pelo resultado do cálculo.

Fonte: Adaptado de Lopes e Garcia (2002, p. 79)

```
1 #obtendo o numero inteiro
2 numero = int(raw_input("Informe um numero inteiro:"))
3
4 #validacao com a clausula if
5 if numero > 20:
6     resul = numero * 2
```

```
7       print "Resultado: ", resul
8   else:
9       resul = numero * 4
10      print "Resultado: ", resul
```

Na resposta apresentada para o exercício, o programa teve que executar a instrução de multiplicação por 2, caso o usuário informasse um valor superior à 20 e a instrução de multiplicação por 4, caso o valor fosse inferior ou igual a 20. Veja que neste caso, um valor igual a 20 foi considerado pela cláusula **else**, isso porque a expressão na condição indica **num > 20**, desta forma, qualquer valor que não seja maior que 20 será processado pela cláusula **else**.

Um ponto chama a atenção na resposta apresentada, ao observar as linhas 7 e 10, chega-se à conclusão de que são iguais, ou seja, embora seja necessário realizar a impressão do resultado nos dois casos, como a variável utilizada e o texto a ser impresso são os mesmos, independente do resultado da condição, então o comando permaneceu inalterado, neste caso, o ideal é que essa instrução seja posicionada fora da estrutura de decisão **if** para evitar redundância de código. Veja a seguir o ajuste feito na resposta.

```
1   resul = 0
2   #obtendo o numero inteiro
3   numero = int(raw_input("Informe um numero inteiro:"))
4
5   #validacao com a clausula if
6   if numero > 20:
7       resul = numero * 2
8   else:
9       resul = numero * 4
10
11  print "Resultado: ", resul
```

Agora note que neste caso, o comando **print** foi movido para fora do bloco de decisão, pois como ele não se altera em relação à condição testada, então este é comum às duas situações e pode ser movido para fora deste fluxo de decisão. Note também que, neste caso, a variável *resul* precisou ser declarada antes do bloco de instruções do **if**, pois como essa variável passou a ser utilizada fora deste escopo, então foi necessário que ela também existisse no escopo geral do programa.

2.6 Cláusula **if**, **elif** e **else** com *n* blocos de instruções

Até o momento foram apresentadas situações que envolvam a execução ou não de um fluxo, neste caso, a presença apenas do comando **if** e a execução de um fluxo ou outro, nesta situação, a presença do **if** e **else**, mas e se for necessário que o programa decida entre 3 fluxos diferentes? Ou ainda, 4 fluxos ou mais? Quando precisar tratar *n* fluxos na decisão das instruções a serem executadas, devemos utilizar o **elif**. Novamente, veja um exercício para exemplificar essa situação.

2.6.1 Exercício de Exemplo

O escritório de contabilidade Super Contábil está realizando o reajuste do salário dos funcionários de todos os seus clientes. Para isso, estão utilizando como referência o reajuste acordado com os sindicatos de cada classe trabalhadora, conforme apresentado na tabela a seguir. Para ajudar o escritório nesta tarefa, faça um programa em Python

que receba o salário atual, o cargo conforme especificado na tabela a seguir e realize o cálculo do reajuste do salário. Ao término do cálculo o programa deve imprimir o valor do reajuste e o novo salário do funcionário.

Cód. cargo	*Cargo*	*% reajuste acordado*
1	Auxiliar de escritório	7%
2	Secretário(a)	9%
3	Cozinheiro(a)	5%
4	Entregador(a)	12%

Para resolver este exercício foi utilizado o código do cargo para determinar a qual cargo pertence o funcionário, no qual o salário está sendo reajustado, assim, foi declarada uma variável do tipo inteiro para armazenar o cargo e foram declaradas duas variáveis do tipo real para armazenar o salário atual do funcionário e o valor do reajuste. Além disso, foi utilizada a estrutura de decisão **if**, **elif** e o **else** para decidir qual fluxo executar de acordo com o cargo do funcionário. Como são 4 cargos, então são necessários 4 fluxos distintos na estrutura de decisão. Veja a seguir a resposta do exercício e os comentários.

```
1  cargo = int(raw_input("Informe o cargo do funcionario:"))
2  salAtual = float(raw_input("Informe o salario atual:"))
3  reajuste = 0
4
5  if cargo == 1:
6      reajuste = (salAtual * 7) / 100
7  elif cargo == 2:
8      reajuste = (salAtual * 9) / 100
9  elif cargo == 3:
10     reajuste = (salAtual * 5) / 100
11 else:
12     reajuste = (salAtual * 12) / 100
13
14 print "O reajuste e: ", reajuste
15 print "O novo salario e: ", salAtual + reajuste
```

A resposta apresentada para o exercício traz uma possível solução para o exemplo. Apesar de mencionar pela primeira vez as palavras "possível solução", o mesmo conceito se aplica aos exercícios anteriores e aos próximos, isso quer dizer que não há uma única solução para os exercícios apresentados, nem mesmo os mais simples exercitados, todos eles possuem soluções diferentes das apresentadas que serão satisfatórias. E dificilmente dois programadores irão produzir soluções iguais para um mesmo problema, é provável que produzam soluções parecidas, mas é bem pouco provável que façam uma resposta igual, mesmo que seja apenas a diferença nos nomes propostos para as variáveis, sempre haverá alguma diferença pelo simples fato de que as pessoas pensam diferente umas das outras e a programação permite construções distintas para um mesmo problema.

Bem, em relação aos comentários da solução proposta, veja que nas linhas 1, 2 e 3 foram declaradas as variáveis que são necessárias no programa, uma variável do tipo inteiro (int) para armazenar o cargo do funcionário, denominada por *cargo*, duas variáveis do tipo real (float) para armazenar o salário atual, *salAtual* e o resultado do cálculo do reajuste a ser aplicado ao salário atual, denominada *reajuste*. O que determinou o tipo das variáveis foi a conversão no momento da leitura do valor pelo usuário. O cargo será utilizado no cálculo do reajuste e o salário atual também.

Na linha 5 do exercício de exemplo é iniciado o fluxo de decisão que considera o *cargo* do funcionário como sendo o fator de decisão para a escolha entre os fluxos, pois é o cargo que determina o reajuste a ser aplicado. Desta forma, a linha 5 traz a seguinte condição: **"cargo == 1"**, note que o teste de igualdade é determinado pelo uso da igualdade dupla, conforme pode ser visto na tabela 6, assim se o usuário informar o código **1** para o cargo, então o programa executará o bloco de instruções do primeiro fluxo, ou seja, a linha 6. Observe como foi feito o cálculo de percentual de reajuste: **"reajuste = (salAtual * 7) / 100"**, nota-se que o salário atual foi multiplicado por 7 conforme o cargo 1 e o resultado dessa multiplicação foi divido por 100 para então, o resultado desta expressão ser atribuído à variável *reajuste*. A presença dos parênteses **"( )"** na primeira parte da expressão matemática, garante que esta será sempre a primeira a ser realizada, assim, caso a operação fosse a soma, por exemplo, ela seria executada primeiro que a segunda parte da expressão, mesmo a segunda parte sendo uma divisão.

Na linha 7, foi incluído o segundo fluxo de decisão e a primeira ocorrência da instrução **elif** para possibilitar que uma nova condição seja testada caso a anterior seja falsa. Esse é o princípio do funcionamento quando são necessários vários blocos de condição, se a primeira retornar falso como resultado, ao invés de apenas executar o próximo bloco, será feito um novo teste de condição, se o próximo teste também retornar falso, será realizado um novo teste no próximo fluxo encontrado e assim por diante, até o último fluxo, em que há apenas a cláusula **else**.

As linhas 11 e 12 trazem o último fluxo, totalizando assim os 4 fluxos necessários para o teste dos cargos. As linhas 14 e 15 possibilitam a conclusão do exercício pela impressão do reajuste a ser aplicado, na linha 14, e do novo salário, na linha 15. Note na linha 15, que por opção, o cálculo do novo salário, ou seja, **"salAtual + reajuste"**, foi realizado no comando **print**, neste caso, o valor do salário novo não foi armazenado em nenhuma variável, foi apenas calculado para impressão.

2.7 Cláusula **if**, **elif** e **else** com condições compostas

Até agora, foi ensinado a utilizar a estrutura de decisão **if** com apenas um bloco, com dois blocos, e com *n* blocos de instruções. Foi ensinado também a utilizar os operadores relacionais apresentados na tabela 6. Agora é necessário aprender a fazer testes de condição com a combinação de operadores lógicos, em que é possível fazer uso de conjunções, disjunções e negação (DOWNEY; ELKNER; MEYERS, 2010, p. 36). Em linguagem Python é possível combinar vários operadores em uma mesma condição, inclusive pode-se também combinar operadores diferentes. Para iniciar, veja primeiro a tabela 7 que apresenta os operadores lógicos. Na sequência veja um exercício para exemplificar o uso da combinação de condições.

Tabela 7 – Operadores lógicos

Operador Lógico	Representação em C	Exemplo
E (conjunção)	and	x > 1 and x < 19
OU (disjunção)	or	x == 1 or x == 2
NÃO (negação)	not	not Continuar

Fonte: Adaptado de Downey, Elkner e Meyers (2010, p. 36)

2.7.1 Exercício de Exemplo

O hospital local está fazendo uma campanha para receber doação de sangue. O propenso doador deve inicialmente se cadastrar informando o seu nome completo, sua idade, seu peso, responder a um breve questionário e apresentar um documento oficial com foto. Faça um programa que permita ao hospital realizar o cadastro dos voluntários para avaliar a aptidão quanto à doação de sangue. Para estar apto a doar, o voluntário deve ter idade entre 16 e 69, pesar pelo menos 50 kg, estar bem alimentado e não estar resfriado. O programa deve ler os dados e imprimir no final o nome do voluntário e se ele está apto ou não.

```
1  nome = raw_input("Informe o nome:")
2  peso = float(raw_input("Informe o peso:"))
3  idade = int(raw_input("Informe a idade:"))
4  bemAlimentado = int(raw_input("Esta bem alimentado? <1-SIM / 0-NAO>"))
5  resfriado = int(raw_input("Esta resfriado? <1-SIM / 0-NAO>"))
6  if peso >= 50 and (idade >= 16 and idade <= 69) and bemAlimentado
7      and (not resfriado):
8      print "O voluntario %s esta apto!" %  nome
9  else:
10     print "O voluntario %s NAO esta apto" % nome
```

Pode-se ver na resposta sugerida para o exercício, que na linha 1 foi feita a declaração e atribuição da variável que será responsável por armazenar o nome do voluntário. Nas linhas 2 a 5 foram declaradas as demais variáveis para armazenar o peso, a idade e os indicativos de bem alimentado e resfriado. No caso dos indicativos, note que as variáveis foram definidas com o tipo **int**, no momento da conversão, desta forma, adotou-se os valores 1 quando afirmativo e 0 quando negativo.

Na linha 6 inicia-se o bloco de decisão. Note que o comando **if** inclui várias condições que são combinadas pelo operador lógico **and** que corresponde ao 'E' (conjunção) conforme a tabela 7. Na figura 3 pode ser visto em detalhes qual trecho do código corresponde à cada condição presente na estrutura de decisão **if**. Veja que a instrução inclui 4 condições, sendo que a segunda condição é composta e por sua vez possui 2 condições, totalizando assim 5 condições.

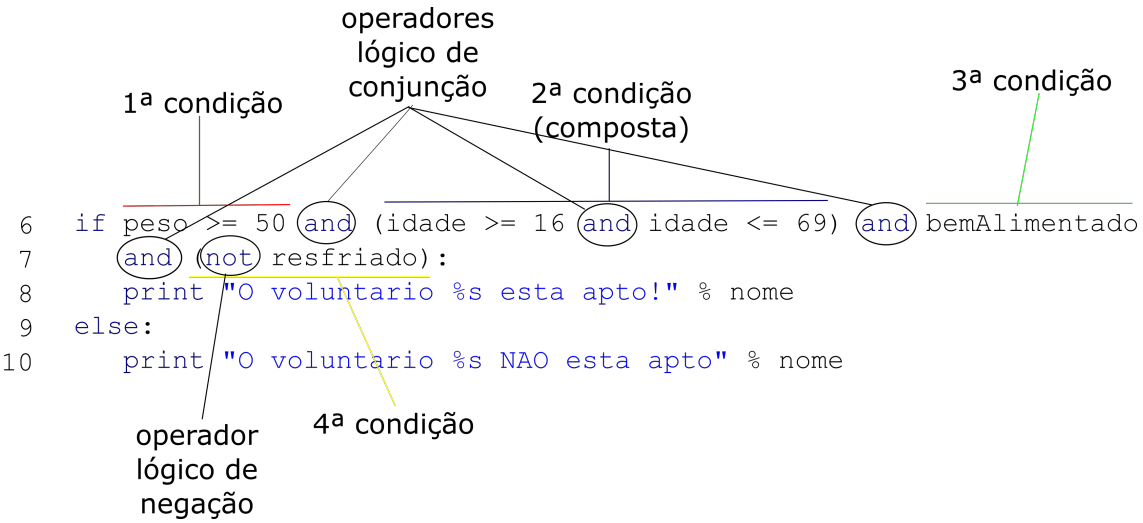

Figura 3 – Exemplo de aplicação dos operadores lógicos

A figura 3 mostra também os operadores lógicos, sendo o operador **and** que é responsável por combinar as condições. O operador de conjunção indica que, para que essa expressão de condição seja considerada verdadeira, todas as condições devem ser verdadeiras, basta que uma delas seja falsa para que a combinação de todas as condições seja considerada falsa. Exemplo, se a idade informada pelo usuário for **15**, toda a expressão de condição será considerada falsa, pois todas devem ser verdadeiras para que o conjunto de condições seja considerado verdadeiro. E também o operador de negação **not**, que neste caso, é utilizado para retornar o contrário do que está armazenado na variável *resfriado*, assim, se o usuário informou, por exemplo, **1** indicando que o voluntário está resfriado, então a negação irá retornar o contrário, **0** (zero), e no teste da condição a expressão será considerada falsa, pois não é aceitável um doador resfriado.

Então, nos testes lógicos em linguagem Python, sempre será **falso** quando o valor inteiro for igual à **0** (zero) e será verdadeiro quando o valor for diferente de zero, ou seja, qualquer outro que não seja zero, incluindo números negativos (DOWNEY; ELKNER; MEYERS, 2010, p. 36). Desta forma, no exercício de exemplo apresentado, é preciso uma pessoa bem alimentada, assim, se o usuário responder **0**, o teste da condição irá considerar falso, mas se o usuário responder qualquer número diferente de zero, então será considerado verdadeiro, isso quer dizer, que apesar do programa orientar ao usuário responder **"1-SIM / 0-NAO"** conforme a linha 15 da solução proposta, se por algum motivo o usuário responder, por exemplo, **2**, será considerado verdadeiro, mesmo que ele tenha respondido um valor fora da faixa disponibilizada pelo programa. Neste caso, o ideal seria forçar o usuário a responder apenas conforme indicado pelo programa, mas isso é assunto para a próxima aula.

Veja agora mais um exercício de exemplo, neste caso, utilizando o operador lógico 'OU' (disjunção).

2.7.2 Exercício de Exemplo

Segundo uma tabela médica, o peso ideal está relacionado com a altura e o sexo. Faça um programa em Python que receba a altura e o sexo de uma pessoa, após isso calcule e imprima o seu peso ideal, utilizando as seguintes fórmulas:

- Para homens: (72,7 * A) – 58
- Para mulheres: (62,1 * A) – 44,7
- Em que:

  A = Altura

Fonte: Adaptado de Lopes e Garcia (2002, p. 101)

```
1 pesoIdeal = 0
2 altura = float(raw_input("Informe a altura:"))
3 sexo = raw_input("Informe o sexo: <M ou F>")
4 if sexo == 'm' or sexo == 'M':
5     pesoIdeal = (72.7 * altura) - 58
6 else:
7     pesoIdeal = (62.1 * altura) - 44.7
8
9 print "O peso ideal e: ", pesoIdeal
```

O objetivo deste exemplo é mostrar o uso do operador lógico de disjunção, o 'OU'. Entre as linhas 1 e 3 foi feita a declaração das variáveis e a leitura dos valores informados pelo usuário, é importante notar que a variável *sexo* não precisou ser convertida, pois o objetivo é que ela seja do tipo **string**, ou seja, para possibilitar ao usuário que informe uma letra. As letras tem variações entre minúscula e maiúscula, neste exemplo, serão utilizadas duas consoantes, 'M' ou 'm' e 'F' ou 'f'. Desta forma, ao informar o sexo, pode ocorrer do usuário informar com letra maiúscula ou minúscula mesmo com a orientação da linha 3.

A estrutura de decisão será responsável então, por tratar essa variação entre letras minúsculas ou maiúsculas. Na linha 4 da resposta proposta são apresentadas então as duas condições para a validade da condição geral, ou seja, se o usuário informar 'M' ou 'm' a condição geral será verdadeira e a instrução para o cálculo do peso ideal será feito para o homem na linha 5. Se o usuário informar qualquer outra letra, então a condição será considerada falsa e será executado o cálculo para o peso ideal da mulher na linha 7. Após realizar o cálculo, o resultado do peso ideal é impresso pela instrução da linha 9 da resposta do exercício.

2.8 Cláusula **if**, **elif** e **else** com condições aninhadas

A estrutura de decisão **if** permite também o uso de condições aninhadas (encaixadas) que são úteis quando é necessário tomar uma decisão dentro de outra (DOWNEY; ELKNER; MEYERS, 2010, p. 38). A seguir um exercício para exemplificar o uso de **if** aninhado.

2.8.1 Exercício de Exemplo

Faça um programa em Python que leia o destino do passageiro, se a viagem inclui retorno (ida e volta) e informe o preço da passagem conforme a tabela a seguir:

CÓD. DESTINO	DESTINO	IDA	IDA E VOLTA
1	Região Norte	500,00	900,00
2	Região Nordeste	350,00	650,00
3	Região Centro-oeste	350,00	600,00
4	Região Sul	300,00	550,00

Fonte: Adaptado de Lopes e Garcia (2002, p. 113)

Para resolver este exercício será necessário primeiro verificar qual é o destino, depois validar se o trecho inclui somente ida ou ida e volta, ou seja, há uma verificação condicionada a outra, a primeira será o destino e a segunda e aninhada à primeira, será a condição do trecho.

```
1  print "Informe o destino conforme tabela a seguir: \n"
2  print "1-Regiao Norte \t 3-Regiao Centro-oeste \n"
3  print "2-Regiao Nordeste \t 4-Regiao Sul \n"
4  destino = int(raw_input("Destino:"))
5  trecho = int(raw_input("Informe o trecho: <1-IDA ou 2-IDA E VOLTA>"))
6  if destino == 1:
7      if trecho == 1:
8          print "Regiao norte[IDA] = 500,00"
9      else:
10         print "Regiao norte[IDA E VOLTA] = 900,00"
11 elif destino == 2:
```

```
12      if trecho == 1:
13          print "Regiao nordeste[IDA] = 350,00"
14      else:
15          print "Regiao nordeste[IDA E VOLTA] = 650,00"
16  elif destino == 3:
17      if trecho == 1:
18          print "Regiao centro-oeste[IDA] = 350,00"
19      else:
20          print "Regiao centro-oeste[IDA E VOLTA] = 600,00"
21  else:
22      if trecho == 1:
23          print "Regiao sul[IDA] = 300,00"
24      else:
25          print "Regiao sul[IDA E VOLTA] = 550,00"
```

Até a linha 5 da resposta apresentada para o exercício, há instruções que permitem ao usuário informar o destino e o trecho conforme a tabela do exercício. A partir da linha 6 foi incluída a estrutura de condição com **if** aninhado. Note que, primeiro é feita a comparação com o destino na linha 6, pois o trecho irá apresentar diferentes valores para cada diferente destino, por isso, é necessário analisar a condição do trecho 'dentro' da condição do destino. Na primeira condição, se o destino escolhido é igual à **1**, ou seja, 'Região Norte', então a instrução na linha 7 será executada, se o trecho escolhido também é igual à **1**, então a instrução na linha 8 será executada, mas se o trecho escolhido é diferente de **1**, então a instrução na linha 10 será executada.

As linhas seguintes seguem o mesmo padrão, mas atendendo aos demais destinos e trechos da tabela do exercício, a linha 21 traz a última condição para destino, que é contrária aos destinos 1, 2 e 3, ou seja, qualquer outro destino diferente destes levarão à execução das instruções a partir da linha 21. Não há um limite para incluir um **if** aninhado, 'dentro', de outro **if** em níveis, é possível incluir quantos níveis forem necessários, contudo, um número de níveis muito grande não é muito prático em termos de programação, principalmente no que diz respeito à manutenção do código-fonte, então é melhor evitar muitos níveis.

2.9 Resumo da Aula

Na aula 2 foram apresentados os conceitos necessários para construir um programa em linguagem Python com fluxos de execução distintos, ou seja, aplicações que executem tarefas ou ações diferentes de acordo com os dados de entrada do programa. Assim, o programa pode por exemplo, decidir efetuar um cálculo em detrimento de outro em função dos dados de entrada, em computação, isso caracteriza uma estrutura de decisão.

Neste sentido, foram apresentadas as cláusulas de estrutura de decisão em linguagem Python, a cláusula **if**, **elif** e **else** em suas variadas possibilidades. Essas cláusulas são poderosas, pois dão várias alternativas ao programador, como estabelecer se um bloco de instruções será ou não executado, ou definir dois blocos de instruções, sendo ao menos um dos dois executados sempre, ou ainda, estabelecer n blocos de instruções. Além disso, essas cláusulas permitem avaliar expressões de igualdade, de diferença, entre outras, e possibilitam também combinar expressões de relação diferentes, por exemplo, combinar uma expressão de igualdade com uma expressão de diferença.

2.10 Exercícios da Aula

Os exercícios desta lista foram Adaptados de Lopes e Garcia (2002, p. 79-120).

1. Faça um programa em Python que leia dois valores numéricos inteiros e efetue a adição, caso o resultado seja maior que 10, apresentá-lo.

2. Faça um programa em Python que leia dois valores inteiros e efetue a adição. Caso o valor somado seja maior que 20, este deverá ser apresentado somando-se a ele mais 8, caso o valor somado seja menor ou igual a 20, este deverá ser apresentado subtraindo-se 5.

3. Faça um programa que leia um número e imprima uma das duas mensagens: "É múltiplo de 3"ou "Não é múltiplo de 3".

4. Faça um programa que leia um número e informe se ele é ou não divisível por 5.

5. Faça um programa em Python que leia um número e informe se ele é divisível por 3 e por 7.

6. A prefeitura do Rio de Janeiro abriu uma linha de crédito para os funcionários estatutários. O valor máximo da prestação não poderá ultrapassar 30% do salário bruto. Faça um programa em linguagem Python que permita entrar com o salário bruto e o valor da prestação e informar se o empréstimo pode ou não ser concedido.

7. Faça um programa em Python que leia um número e indique se o número está compreendido entre 20 e 50 ou não.

8. Faça um programa que leia um número e imprima uma das mensagens: "Maior do que 20", "Igual a 20"ou "Menor do que 20".

9. Faça um programa em Python que permita entrar com o ano de nascimento da pessoa e com o ano atual. O programa deve imprimir a idade da pessoa. Não se esqueça de verificar se o ano de nascimento informado é válido.

10. Faça um programa em Python que leia três números inteiros e imprima os três em ordem crescente.

11. Faça um programa que leia 3 números e imprima o maior deles.

12. Faça um programa que leia a idade de uma pessoa e informe:

    - Se é maior de idade
    - Se é menor de idade
    - Se é maior de 65 anos

13. Faça um programa que permita entrar com o nome, a nota da prova 1 e a nota da prova 2 de um aluno. O programa deve imprimir o nome, a nota da prova 1, a nota da prova 2, a média das notas e uma das mensagens: "Aprovado", "Reprovado"ou "em Prova Final"(a média é 7 para aprovação, menor que 3 para reprovação e as demais em prova final).

14. Faça um programa que permita entrar com o salário de uma pessoa e imprima o desconto do INSS segundo a tabela seguir:

Salário	Faixa de Desconto
Menor ou igual à R$600,00	Isento
Maior que R$600,00 e menor ou igual a R$1200,00	20%
Maior que R$1200,00 e menor ou igual a R$2000,00	25%
Maior que R$2000,00	30%

15. Um comerciante comprou um produto e quer vendê-lo com um lucro de 45% se o valor da compra for menor que R$20,00, caso contrário, o lucro será de 30%. Faça um programa em Python que leia o valor do produto e imprima o valor da venda.

16. A confederação brasileira de natação irá promover eliminatórias para o próximo mundial. Faça um programa em Python que receba a idade de um nadador e imprima a sua categoria segundo a tabela a seguir:

Categoria	Idade
Infantil A	5 - 7 anos
Infantil B	8 - 10 anos
Juvenil A	11 - 13 anos
Juvenil B	14 - 17 anos
Sênior	maiores de 18 anos

17. Depois da liberação do governo para as mensalidades dos planos de saúde, as pessoas começaram a fazer pesquisas para descobrir um bom plano, não muito caro. Um vendedor de um plano de saúde apresentou a tabela a seguir. Faça um programa que entre com o nome e a idade de uma pessoa e imprima o nome e o valor que ela deverá pagar.

Idade	Valor
Até 10 anos	R$30,00
Acima de 10 até 29 anos	R$60,00
Acima de 29 até 45 anos	R$120,00
Acima de 45 até 59 anos	R$150,00
Acima de 59 até 65 anos	R$250,00
Maior que 65 anos	R$400,00

18. Faça um programa que leia um número inteiro entre 1 e 12 e escreva o mês correspondente. Caso o usuário digite um número fora desse intervalo, deverá aparecer uma mensagem informando que não existe mês com este número. Utilize o **switch** para este problema.

19. Em um campeonato nacional de arco-e-flecha, tem-se equipes de três jogadores para cada estado. Sabendo-se que os arqueiros de uma equipe não obtiveram o mesmo número de pontos, criar um programa em Python que informe se uma equipe foi classificada, de acordo com a seguinte especificação:

    - Ler os pontos obtidos por cada jogador da equipe;
    - Mostrar esses valores em ordem decrescente;
    - Se a soma dos pontos for maior do que 100, imprimir a média aritmética entre eles, caso contrário, imprimir a mensagem "Equipe desclassificada".

20. O banco XXX concederá um crédito especial com juros de 2% aos seus clientes de acordo com o saldo médio no último ano. Faça um programa que leia o saldo médio de um cliente e calcule o valor do crédito de acordo com a tabela a seguir. O programa deve imprimir uma mensagem informando o saldo médio e o valor de crédito.

Saldo Médio	**Percentual**
de 0 a 500	nenhum crédito
de 501 a 1000	30% do valor do saldo médio
de 1001 a 3000	40% do valor do saldo médio
acima de 3001	50% do valor do saldo médio

21. A biblioteca de uma Universidade deseja fazer um programa que leia o nome do livro que será emprestado, o tipo de usuário (professor ou aluno) e possa imprimir um recibo conforme mostrado a seguir. Considerar que o professor tem dez dias para devolver o livro e o aluno só três dias.

    - Nome do livro:
    - Tipo de usuário:
    - Total de dias:

22. Construa um programa que leia o percurso em quilómetros, o tipo do carro e informe o consumo estimado de combustível, sabendo-se que um carro tipo Python faz 12 km com um litro de gasolina, um tipo B faz 9 km e o tipo C, 8 km por litro.

23. Crie um programa que informe a quantidade total de calorias de uma refeição a partir da escolha do usuário que deverá informar o prato, a sobremesa, e bebida conforme a tabela a seguir.

Prato	**Sobremesa**	**Bebida**
Vegetariano 180cal	Abacaxi 75cal	Chá 20cal
Peixe 230cal	Sorvete diet 110cal	Suco de laranja 70cal
Frango 250cal	Mousse diet 170cal	Suco de melão 100cal
Carne 350cal	Mousse chocolate 200cal	Refrigerante diet 65cal

24. A polícia rodoviária resolveu fazer cumprir a lei e vistoriar veículos para cobrar dos motoristas o DUT. Sabendo-se que o mês em que o emplacamento do carro deve ser renovado é determinado pelo último número da placa do mesmo, faça um programa que, a partir da leitura da placa do carro, informe o mês em que o emplacamento deve ser renovado.

25. A prefeitura contratou uma firma especializada para manter os níveis de poluição considerados ideais para um país do 1º mundo. As indústrias, maiores responsáveis pela poluição, foram classificadas em três grupos. Sabendo-se que a escala utilizada varia de 0,05 e que o índice de poluição aceitável é até 0,25, fazer um programa que possa imprimir intimações de acordo com o índice e a tabela a seguir:

Índice	Indústrias que receberão intimação
0,3	1º gurpo
0,4	1º e 2º grupos
0,5	1º, 2º e 3º grupos

AULA 3

Estruturas de Iteração

Metas da Aula

1. Entender e praticar os conceitos da sintaxe utilizada na linguagem Python para estruturas de iteração.
2. Aplicar variadas situações relacionadas ao fluxo de iteração em programação com o objetivo de cobrir diversificadas possibilidades vivenciadas na programação cotidiana.
3. Escrever programas que farão uso de estruturas de iteração.

Ao término desta aula, você será capaz de:

1. Escrever programas em linguagem Python que sejam capazes de resolver problemas que envolvam, por exemplo, a repetição de várias instruções em função de uma determinada condição.
2. Escrever programas que manipulem informações repetidas vezes considerando variadas situações e condições.

3.1 Estruturas de Iteração

Até esta parte do livro foram trabalhados conceitos que nos permitem escrever programas em linguagem Python que sejam capazes de realizar operações básicas, como: operações matemáticas, operações com a memória e com os dispositivos de entrada e saída. Além disso, lidar com fluxos diferentes dada uma determinada condição ou condições. Isso nos permite então cobrir parte dos problemas computacionais que surgem, mas ainda há situações que devem ser tratadas, dentre elas, a possibilidade do programa executar um número *n* de instruções. Exemplo, um dos exercícios realizados na aula 2 tem o objetivo de calcular o reajuste do salário de um funcionário conforme o seu cargo, na resolução do exercício é apresentado um programa que executa o cálculo para 1 funcionário e encerra, bem, imagine então uma situação hipotética em que a empresa tenha que fazer o cálculo do reajuste para 50 funcionários, neste caso ela terá que executar o programa 50 vezes manualmente, pois o programa apresentado como solução não é capaz de automatizar essa operação para todos os funcionários.

Então, para resolver esse problema computacional, é necessário lançar mão das estruturas de iteração que irão permitir que o nosso programa Python tenha essa capacidade. Nesta aula serão apresentadas duas estruturas de iteração, a estrutura **while** e o **for**.

3.2 Cláusula for

A cláusula **for** é muito útil quando se deseja repetir uma ou várias instruções por um número *n* de vezes ou iterar em uma lista de dados. Diferente de outras linguagens, no Python, o uso mais comum do **for** é percorrendo listas, mas também é possível fazer o seu uso iterando por um determinado número de vezes (COELHO, 2007, p. 30). Veja a seguir a sintaxe do **for**:

```
1 for objeto in lista:
2     instrucao
3 else:
4     instrucao
```

Na linha 1 da sintaxe apresentada foi incluída a palavra reservada **for** para indicar o início do laço, na sequência a palavra **objeto** indica o que será o item que vai iterar na lista, e a **lista** é a que se deseja percorrer, que pode ser uma lista realmente, ou um número de iterações. É com base nestas definições que o comando irá definir quantas iterações serão realizadas. Na linha 2 foi adicionada a instrução que será executada *n* vezes. O **for** segue o mesmo padrão de outros comandos como o **if** que podem incluir blocos de instrução, ou seja, executar várias instruções, para isso, basta indicar com a indentação quais instruções serão executadas dentro do laço (COELHO, 2007; LUTZ; ASCHER, 2007; CRUZ, 2017).

Algo super interessante no laço **for** do Python é que ele aceita a instrução **else**, como pode-se ver na linha 3 da sintaxe. O **else** no **for** é sempre acionado caso não ocorra nenhuma interrupção ao longo do ciclo do laço, ou seja, caso todos os itens ou objetos da lista sejam visitados (LUTZ; ASCHER, 2007, p. 178). Assim, podemos utilizar o **else** quando desejarmos executar *n* instruções após toda a lista ser visitada. Mas, é importante salientar que o **else** é opcional.

Uma das formas de se interromper as iterações do laço **for** é pelo comando **break**. Ao acionar este comando no bloco de instruções do **for** ele imediatamente abandona o laço interrompendo as próximas iterações que seriam executadas.

Outro conceito importante a ser explicado, é do **range()**. Mas, o que é isso? Bem, na linguagem Python o **for** foi "desenhado"para trabalhar com objetos, assim, realizar iterações por um determinado número de vezes, como é comum em outras linguagens, não seria o uso natural em Python. Mas é possível, por meio do **range()** (CRUZ, 2017, p. 39). O **range()** retorna um objeto iterável, que embora não seja uma lista, é compatível com o **for** e nos permite determinar o número de iterações do laço, sem termos uma lista preenchida. A seguir um exemplo em forma de exercício para iniciarmos com os conceitos do **for**.

3.2.1 Exercício de Exemplo

Faça um programa em Python que leia 10 valores e ao final imprima a média aritmética dos valores lidos.

Fonte: Adaptado de Lopes e Garcia (2002, p. 152)

```
1  soma = 0
2  media = 0
3  #inicio do laco for
4  for i in range(10):
5      #a partir deste ponto sao as instrucoes
6      #que devem ser executadas nas iteracoes
7      num = float(raw_input("Informe o numero:"))
8      soma += num
9
10 #a media deve ser calculada apos o laco
11 media = soma / 10
12 print "A media e: ", media
```

Nas linhas 1 e 2 da resposta do exercício foram declaradas as variáveis *soma* para acumular os valores que são inseridos pelo usuário e a variável *media* para armazenar o resultado do cálculo da média aritmética. A cláusula **for** inicia na linha 4 em que são feitas as definições do laço. A primeira definição é a inicialização da variável *i*, visto que não teremos um objeto, então ela fará o papel de armazenar o número da iteração, a cada iteração. A segunda definição, **range(10)**, indica que as iterações serão executadas até que o **range()** retorne esse valor.

Entre as linhas 5 e 8 foram adicionadas as instruções que serão executadas pelo laço **for**, que no caso são: Linha 7, leitura do valor informado pelo usuário, como o programa deve ler 10 valores, essa instrução deve ser incluída no laço, linha 8, soma dos valores lidos. As linhas 5 e 6 são apenas comentários. O objetivo de somar dentro do laço está associado à fórmula da média aritmética, pois, para calcular essa média é necessário somar primeiro todos os valores e depois dividir pela quantidade total dos valores. Desta forma, essa instrução deve ser incluída no laço, pois 10 valores deverão ser somados.

A linha 11, traz a instrução responsável por dividir o total somado dos valores pela quantidade total de valores, que no caso é 10, note que, essa instrução será executada fora do laço, pois ela está recuada ao nível do programa principal. A instrução na linha 12 é responsável pela impressão do resultado final. O exercício de exemplo, mostra claramente como é possível utilizar o **for** para executar várias iterações de uma ou várias rotinas. O exercício de exemplo pode nos levar a uma questão: e se não é conhecido previamente o número de iterações necessárias? Neste caso, pode-se utilizar uma variável para determinar o número total de iterações. Para exemplificar, veja a seguir um exercício ajustado em relação ao exemplo anterior.

3.2.2 Exercício de Exemplo

Faça um programa em Python que leia *n* valores. O programa deve inicialmente solicitar ao usuário que informe a quantidade desejada de valores a ser informada, depois ler os *n* valores e ao final imprimir a média aritmética dos valores lidos.

```
1  soma = 0
2  media = 0
3  qtdeNum = int(raw_input("Informe a quantidade de numeros:"))
4  #inicio do laco for
5  for i in range(qtdeNum):
6      #a partir deste ponto sao as instrucoes
7      #que devem ser executadas nas iteracoes
8      num = float(raw_input("Informe o numero:"))
9      soma += num
10
11 #a media deve ser calculada apos o laco
12 media = soma / qtdeNum
13 print "A media e: ", media
```

Note na resposta do exercício que foram necessárias apenas poucas alterações para que o programa atenda à nova necessidade. Primeiro, na linha 3 foi adicionada a declaração da variável *qtdeNum* e foi lido o total de números que devem ser informados no laço. A linha 5, com a cláusula **for** foi ajustada de forma que o valor 10 foi substituído pelo nome da variável *qtdeNum*, assim, ao invés do **range()** receber um valor fixo, ele receberá um valor que poderá variar conforme o que o usuário digitar. Outra linha ajustada é a 12, em que o valor 10 foi novamente substituído pelo nome da variável que controla a quantidade, pois como não é conhecida a quantidade, a priori, este valor também deixa de ser fixo no cálculo da média.

É possível combinar laços com outras estruturas, como estrutura de decisão, desta forma, pode-se, por exemplo, combinar o uso do **for** com o **if**. A seguir um exercício para exemplificar.

3.2.3 Exercício de Exemplo

Faça um programa em Python que imprima todos os valores pares entre 1 e 20.

```
1  for i in range(20):
2      if i % 2 == 0:
3          print "Numero par: %d \n" % i
```

Na linha 1 da resposta proposta para o exercício, foram feitas as definições do laço **for**, com número de iterações igual a 20. Como o exercício requer a impressão apenas dos números pares, então, na linha 2 foi incluída a estrutura de decisão **if** dentro do bloco de instruções do **for**, ou seja, o **if** está aninhado ao **for**. Embora o **if** utilizado no exemplo está em sua forma mais simples, pode-se combinar o **for** com *n* estruturas em sua forma mais complexa.

3.3 Cláusula for com laços aninhados

A estrutura de iteração **for** permite também o uso de laços aninhadas (encaixadas) que são úteis quando são necessárias iterações dentro de outras. Não há limite de laços que pode-se aninhar a outros, contudo, quanto mais aninhamentos forem criados, menor

será o desempenho do nosso algoritmo. O gráfico na figura 4 mostra a relação entre número de instruções que são executadas por número de laços aninhados para o caso em que o número de iterações do primeiro laço é 10.

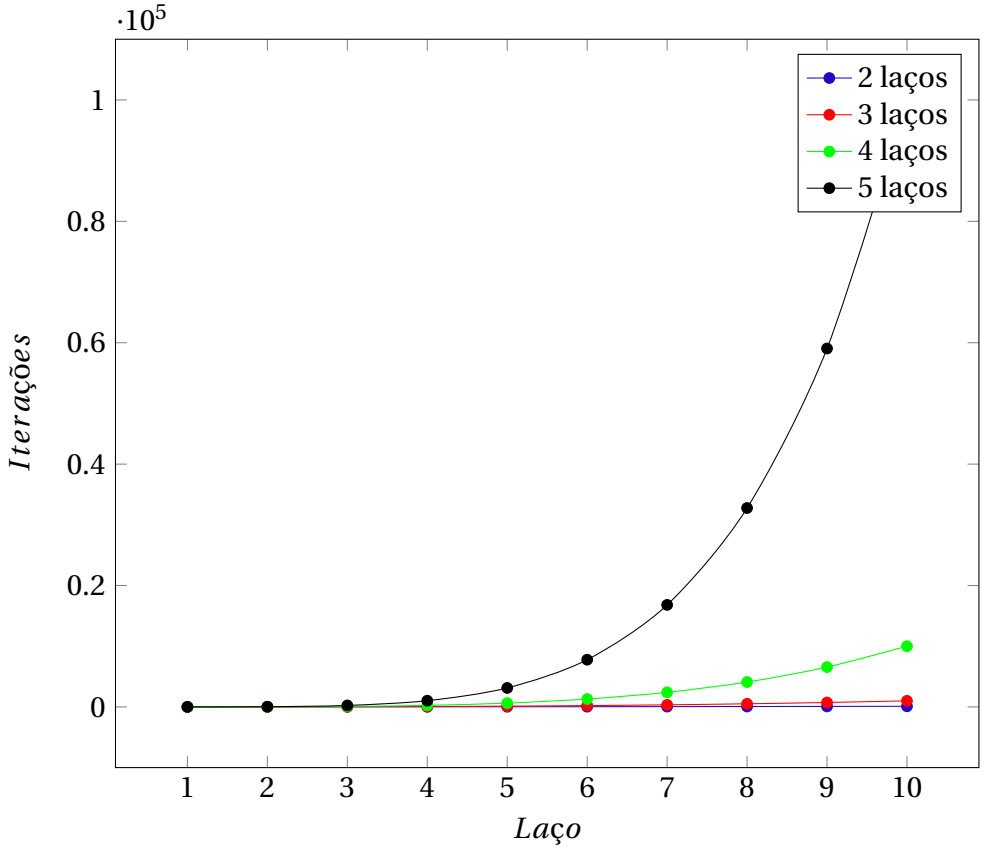

Figura 4 – Nº de instruções executados x Laços aninhados

Veja na figura 4 que a partir de 4 laços aninhados o número de instruções cresce muito, isso porque o aumento é exponencial. Desta forma, ao escrever algoritmos em que haverá muitos laços aninhados, deve-se ter um cuidado especial neste caso, por exemplo, fazer testes suficientes para se certificar de que o tempo de execução será viável. É importante comentar também que, o gráfico mostra o número de instruções que serão executadas para cada caso, assumindo 10 iterações na primeira estrutura de iteração, contudo, o tempo de execução de cada instrução irá variar de acordo com o hardware utilizado. Outro ponto importante a ser destacado, é que, essa relação apresentada na figura 4 vale para qualquer estrutura de iteração, como: **for** e **while**. A seguir mais um exercício para exemplificar o uso de **for** aninhado.

3.3.1 Exercício de Exemplo

Faça um programa em Python que imprima uma matriz de 4 linhas por 4 colunas, sendo que na primeira linha devem ser impressos os valores de 1 à 4 e partir da segunda linha, os valores impressos devem ser múltiplos da linha anterior.

```
1 #inicio do laco do primeiro for
2 for i in range(1, 5):
3     #inicio do laco do segundo for
```

```
4       for j in range(1, 5):
5           if j < 4:
6               print "%d\t" % (j*i),
7           else:
8               print "%d" % (j*i)
```

Saída do Programa

1	2	3	4
2	4	6	8
3	6	9	12
4	8	12	16

Figura 5 – Saída de exemplo de laço aninhado com **for**

Conforme pode ser visto na solução proposta para o exercício, embora aparentemente complexo, com o uso de laços aninhados é muito fácil resolver o exercício. O primeiro laço **for** na linha 2 é responsável pelas iterações nas linhas da matriz e o laço na linha 4 é responsável pela iteração nas colunas. Notou que há algo diferente no uso do **range**()? Incluímos um parâmetro adicional para indicar qual o valor inicial do **range**(), no caso **1**. Note ainda que, como há dois laços aninhados, foi necessário utilizar duas variáveis de controle, pois caso fosse utilizada a mesma variável para controle dos dois laços ocorreriam problemas, pois um laço iria alterar o valor da variável de iteração do outro laço, isso iria provocar efeitos inesperadas nos laços, assim, é sempre comum utilizar uma variável de controle para cada laço aninhado.

Na linha 5 utilizou-se um **if** para verificar se a última coluna da linha ainda não foi impressa, "j < 4", pois neste caso, o **print** é combinado com o código "\t", pois assim, o valor é impresso e uma tabulação é adicionada, caso contrário o **print** vai executar a ação padrão de quebrar a linha. Por fim, a multiplicação do *j* pelo *i* irá produzir exatamente os valores solicitados no exercício, pois a variável *j* sempre terá valores entre 1 e 4, uma vez que, a cada iteração do primeiro laço, o valor de j é reiniciado para 1, na primeira iteração de *i*, **j * i** irá resultar no próprio valor de *j*, pois *i* é 1, a partir da segunda iteração esse valor vai ser sempre múltiplo da linha anterior, uma vez que o valor é resultante de uma multiplicação. Na figura 5 pode ser visto o resultado da saída do programa.

3.4 Cláusula while

Diferente do **for**, o **while** geralmente é empregado quando o número de laços não pode ser determinado com algum fator, como um número de iterações ou os itens de uma lista. A condição do **while** é definida de forma muito similar à definição da condição no **if**. A diferença é que no **if** o objetivo é desviar o caminho de execução para um fluxo de instruções ou outro, no **while** o objetivo será manter a execução de um bloco de instruções em execução, assim como no **for** (LUTZ; ASCHER, 2007, p. 174). Veja a seguir a sintaxe **while**:

```
1 while condicao de laco ou parada:
2     instrucao
3     if condicao: break
```

```
4       if condicao: continue
5   else:
6       instrucao
```

Na linha 1 da sintaxe apresentada foi incluída a palavra reservada **while** para indicar o início do laço, na sequência há a condição de laço ou parada, é com base nesta definição que o comando irá definir quantas iterações serão realizadas. Na linha 2 foi adicionada a instrução que será executada *n* vezes. O **while** segue o mesmo padrão de outros comandos como o **if** ou o **for** que podem incluir blocos de instrução, ou seja, executar várias operações, para isso, basta indicar por meio de indentação. Os comandos **break** e **continue** podem ser incluídos em qualquer ponto do bloco de instruções do **while**.

O **while** avalia a condição definida em seu argumento, se o resultado da avaliação for falso, ou seja, retornar zero, então o laço termina e o programa continua na instrução seguinte ao **while**, se o resultado da avaliação da condição é verdadeira, ou seja, diferente de zero, então os comandos do bloco de instruções do **while** são executadas, assim, enquanto a condição permanecer verdadeira, as operações serão realizadas. Veja um exemplo em forma de exercício para entender melhor.

3.4.1 Exercício de Exemplo

Faça um programa em Python que realize a soma de todos os valores inteiros de 1 a *n*, sendo que *n* deve ser informado pelo usuário.

```
1  soma = 0
2  i = 1
3  n = int(raw_input("Informe o numero n:"))
4  #inicio do laco while
5  while i <= n:
6      soma += i
7      i += 1
8
9  print "Soma: ", soma
```

O exercício mostra um exemplo claro em que o número de iterações não pode ser determinado antes da execução do programa, pois a quantidade de iterações é definida por *n* que deve ser informado pelo usuário, nesta situação é comum utilizar o **while** como estrutura de iteração. Veja que na linha 3 o usuário é solicitado à informar o número "n", pois conforme indicado no exercício, o *n* é o dado de entrada do programa. Na linha 5 é iniciado o **while**, cuja condição que manterá o laço ativo, **i <= n**, assim, se o usuário, por exemplo, informar o valor 5 para *n*, então o laço irá de 1 até 5, uma vez que *i* inicia com 1, conforme a linha 2. No caso do **while**, em geral, é preciso inicializar as variáveis responsáveis pelo controle do laço, por isso a variável *i* foi inicializada com 1 e a variável *n* foi inicializada pelo usuário.

O **while** verificará se a condição na linha 5 é verdadeira, ou seja, se *i* é menor que *n*, caso seja verdadeira então ele executará o bloco de instruções nas linhas 6 e 7. A linha 6 é responsável pela soma de todos os valores entre 1 e *n*, e a linha 7 é responsável pelo incremento de *i*. Este ponto é importante destacar, note que no **for** o incremento da variável está presente entre os argumentos da cláusula **for**, isso porque no caso do **for**, sempre haverá o incremento de um contador, mas no caso do **while**, isso nem sempre é verdade, assim, é necessário fazer o incremento da variável no bloco de instruções do **while**. A linha 9, que é responsável pela impressão do resultado final foi colocada após

o bloco de instruções do **while**, pois como o objetivo é somar todos os valores de 1 a *n*, então somente após a execução de todo o **while** é que a variável *soma* terá acumulado todos os valores.

Uma característica do **while** é a possibilidade de utilizar condições compostas assim como no **if**. O formato de uso segue o mesmo padrão do **if** requerendo o uso dos operadores lógicos de conjunção, disjunção e negação apresentados na tabela 7. As condições compostas são muito úteis em situações em que for necessário que o **while** avalie mais de uma condição para garantir a continuidade ou parada do laço. Não há limite para as condições a serem avaliadas, ou seja, pode-se incluir 2, 3, 4 ou *n* condições. Veja a seguir um exemplo de condição composta em **while**.

3.4.2 Exercício de Exemplo

Uma agência bancária de uma cidade do interior tem, no máximo, 10 mil clientes. Criar um programa em Python que possa entrar com o número da conta, o nome e o saldo de cada cliente. Imprimir todas as contas, os respectivos saldos e uma das mensagens: positivo / negativo. A digitação acaba quando se digita -999 para número da conta ou quando chegar a 10 mil clientes. Ao final, deverá sair o total de clientes com saldo negativo, o total de clientes da agência e o saldo da agência.

Fonte: Adaptado de Lopes e Garcia (2002, p. 197)

```
1  cTotNeg = 0
2  cTot = 0
3  soma=0
4  conta = int(raw_input("Digite o numero da conta ou -999 para terminar:"))
5  #inicio do while
6  while conta > 0 and cTot < 10000:
7      cTot += 1
8      nome = raw_input("Nome:")
9      saldo = float(raw_input("Saldo:"))
10     soma += saldo
11     if saldo < 0:
12         cTotNeg += 1
13         print "%d - %f - negativo" % (conta, saldo)
14     else:
15         print "%d - %f - positivo" % (conta, saldo)
16
17     #le a conta novamente, para determinar se ira ou nao continuar o laco
18     conta = int(raw_input("Digite o numero da conta ou -999 para terminar:"))
19
20 print "\nTotal de clientes com saldo negativo: ", cTotNeg
21 print "\nTotal de clientes da agencia: ", cTot
```

A resposta proposta para o exercício traz entre as linhas 1 e 4 a declaração das variáveis e a inicialização que se faz necessária. A linha 4 é responsável pela leitura do valor da primeira conta a ser registrada, note que essa leitura é realizada antes de inicializar o **while**, pois como esta variável é utilizada na condição do **while** é necessário inicializar ela com algum valor, pois caso contrário, o programa não executará o laço.

O ponto alto do exercício é na inicialização do **while** na linha 6, note que foram incluídas duas condições associadas pela conjunção **E (and)**, como há uma conjunção, então as duas condições devem ser verdadeiras para que o bloco de instruções do **while** seja executado. A primeira condição, **conta > 0**, validará se o número informado para a conta é maior que zero, essa condição tem dois objetivos, o primeiro é garantir

que valores válidos serão informados como número de conta, o segundo objetivo é oferecer à possibilidade do usuário encerrar a digitação antes do total de 10 mil contas, pois ao informar **-999**, esta condição será falsa forçando a parada do laço. A segunda condição, **cTot < 10000**, validará se o total de contas digitadas não atingiu o limite de 10 mil conforme solicitado no exercício.

As linhas seguintes, entre 7 a 18, são responsáveis pela leitura do nome do titular da conta, o saldo, a soma de todos os saldos e a impressão dos dados da conta com a informação de saldo positivo ou negativo, veja que entre as linhas 11 e 15 foi adicionado o uso do **if** no bloco do **while**, ou seja, assim como no **for**, é possível fazer uso de outras estruturas no bloco de instruções do **while** também. É importante notar a linha 18, veja que é idêntica à 4, porque? Bem, como dito antes, a linha 4 está antes da cláusula **while** em função da necessidade de inicializar a variável *conta*, contudo, a cada iteração do **while** é necessário novamente verificar se o usuário deseja continuar digitando novas contas ou se ele quer encerrar, por isso as linhas são repetidas no bloco de instruções do laço. Por fim, as linhas 20 e 21 são responsáveis pela impressão final dos resultados do programa. É importante observar o fato de que a impressão deve ser realizada após o bloco de instruções do **while**, pois caso contrário o programa iria imprimir a cada iteração os valores acumulados nestas variáveis, o que não é desejado neste exercício.

3.5 Validação de dados com while

Na aula 2, estruturas de decisão, ocorreu uma situação em um determinado exercício que será discutida nesta aula. A situação envolvia o preenchimento de um valor dentro de uma faixa de valores, **1** para **Sim** e **0** para **Não**, para ser mais exato. Na ocasião a resolução proposta não incluía uma validação que forçasse o usuário a preencher apenas os valores dentro da faixa disponibilizada. Com os recursos aprendidos até agora, é possível resolver este problema incluindo essa validação com o **while**. A seguir um exemplo em formato de exercício.

3.5.1 Exercício de Exemplo

Faça um programa em linguagem Python que leia 10 números positivos e imprima o quadrado de cada número. Para cada entrada de dados deverá haver um trecho de validação para que um número negativo não seja aceito pelo programa.
Fonte: Adaptado de Lopes e Garcia (2002, p. 192)

```
1 for i in range(10):
2     num = float(raw_input("Informe um numero:"))
3     while num <= 0:
4         num = float(raw_input("\nATENCAO! Informe um numero maior que zero:"))
5
6     print "Quadrado: ", (num * num)
```

No exercício o ponto que nos interessa é a validação que garanta que os números digitados não sejam negativos. Na linha 2 foi incluída a declaração e leitura da variável *num* para o armazenamento do número digitado pelo usuário. Na linha 1 foi adicionado o início do **for** e as definições necessárias para 10 iterações conforme solicitado no exercício. Entre as linhas 3 e 4, foi feita a validação do número lido, veja que o **while** será executado enquanto o número for menor ou igual à zero, **num <= 0**, ou seja, se o usuário digitar um número maior que zero, a análise da condição irá retornar falso e o bloco de instruções do **while** não irá executar, caso o usuário digite um número menor

ou igual a zero, então a linha 4 é executada e, portanto, o usuário será alertado que ele deve digitar um número maior que zero. Na sequência, se o usuário digitar novamente um número inferior ou igual a zero, a condição do **while** será novamente verdadeira e o bloco de comandos será novamente executado até que ele digite um número positivo ou encerre o programa. Por fim, após digitar um número positivo, a cláusula **while** será abortada e a linha 6 será executada, imprimindo o quadrado do número lido.

3.6 Cláusula while com laços aninhados

Assim como na cláusula **for**, o **while** também permite o uso de **while** aninhado, ou seja, um laço **while** dentro de outro laço **while** ou mesmo outro laço **for** (para o **for**, vale a mesma regra, pode-se colocar um laço **while** no bloco de instruções do **for**), na verdade, não há limite para o número de laços que podem ser incluídos dentro de outro laço com aninhamento, contudo, vale a mesma regra apresentada na figura 4, desta forma, quanto maior for o número de laços **while** aninhados, menor será o desempenho do algoritmo, pois o aumento do número de instruções a serem executadas é exponencial. Assim, vale mais uma vez a regra de que deve-se ter cuidado ao usar este recurso. Analise a seguir, um exemplo de laço **while** aninhado por meio de um exercício.

3.6.1 Exercício de Exemplo

Na Usina de Angra dos Reis, os técnicos analisam a perda de massa de um material radioativo. Sabendo-se que este perde 25% de sua massa a cada 30 segundos, criar um programa em Python que imprima o tempo necessário para que a massa deste material se torne menor que 0,10 grama. O programa deve calcular o tempo para várias massas.
Fonte: Adaptado de Lopes e Garcia (2002, p. 204)

```
1  conTempo = 0
2  resp = raw_input("Digite S se desejar novo calculo ou qualquer letra para
       terminar:")
3  #primeiro while
4  while resp == "S" or resp == "s":
5      #lendo a massa
6      massa = float(raw_input("Digite a massa em gramas do material:"))
7      #segundo while que calcula a perda de massa
8      while massa >= 0.10:
9          conTempo += 1
10         massa *= 0.75
11
12     tempo = (conTempo * 30) / 60
13     print "O tempo foi de: %f minutos. \n" % tempo
14     resp = raw_input("Digite S se desejar novo calculo ou qualquer letra para
           terminar:")
```

Para resolver o exercício de exemplo foi necessário declarar 4 variáveis, *conTempo*, responsável por contabilizar o tempo gasto para a perda de massa, foi necessário inicializar essa variável com zero, pois ela vai ser incrementada em um laço **while**. Foram declaradas também as variáveis *massa* para receber e controlar a perda da massa e *tempo* para receber o resultado do cálculo do tempo. Por fim, foi declarada a variável *resp* para controlar a continuação da digitação da massa pelo usuário. As linhas 2 e 14 são responsáveis pela leitura da variável *resp*, o usuário deve informar **S** caso ele queira realizar o cálculo de perda de massa ou qualquer letra caso ele não queira, isso porque

na linha 4 a condição para o **while** executar a iteração é **resp == "S"** ou **resp == "s"**, veja que neste caso foi necessário utilizar uma disjunção **OU (or)** para a condição composta na cláusula **while**. Essa condição composta é importante, pois o usuário pode vir a digitar a letra **S** em formato minúsculo o que resultaria em falso em uma comparação de igualdade com **S** em letra maiúsculo.

Ao validar como verdadeira a condição do **while** na linha 4 o bloco de instruções entre as linhas 5 e 14 será executado. Neste bloco de instruções, foi realizada a leitura da variável *massa* na linha 6, e na linha 8 o que espera-se exemplificar neste exercício, o **while** aninhado ao outro **while** da linha 4, este **while** interno realiza a contagem do tempo para a perda da massa até o alvo que é 0,10 gramas, por isso a condição do while é **massa >= 0.10**, ou seja, enquanto a massa é superior à 0,10 ela precisa incrementar o tempo gasto na linha 9 e sofrer perda de 25% na linha 10. Desta forma, há um laço aninhado com outro, o primeiro laço irá executar enquanto o usuário desejar realizar mais cálculos de perda de massa e o segundo laço será realizado, para cada cálculo de perda de massa, enquanto a perda não atinge o alvo de 0,10 gramas. Por fim, a linha 12 é responsável pela conversão do tempo em segundos, a linha 13 é responsável pela impressão do tempo calculado e convertido em segundos e a linha 14 é responsável pela nova leitura da variável *resp* para que o usuário informe se deseja realizar mais um cálculo.

3.7 do-while com while

A cláusula **do-while**, comum na maioria das linguagens de programação, é diferente do **while** em um detalhe apenas. O bloco de instruções do **do-while** sempre executará ao menos uma vez, pois a condição no **do-while** é avaliada após a execução, assim, mesmo que a condição, após avaliada, seja falsa, o bloco já terá sido executado, desta forma, a condição falsa apenas impedirá uma nova execução do bloco de instruções. Assim, o **do-while** é recomendado quando não é conhecido previamente o número de iterações necessárias e será necessária a execução de pelo menos 1 iteração. Contudo, não há um **do-while** explicito no Python, talvez porque seja muito fácil implementar um 'do-while' no Python, utilizando o **while**. Veja a seguir um exemplo em formato de exercício.

3.7.1 Exercício de Exemplo

Faça um programa em linguagem Python que permita entrar com números e imprimir o quadrado de cada número digitado até entrar um número múltiplo de 6 que deverá ter seu quadrado impresso também.
Fonte: Adaptado de Lopes e Garcia (2002, p. 189)

```
1 while 1:
2     num = int(raw_input("\nDigite um numero ou multiplo de 6 para encerrar:"))
3     print "Quadrado: ", (num ** 2)
4     if (num % 6) != 0:
5         continue
6     else:
7         break
```

O exercício pede que todos os valores lidos tenham o seu quadrado impresso, e define o ponto de parada do laço como sendo um número lido múltiplo de 6, contudo, como todos devem ser impressos, inclusive o múltiplo de 6, então há uma situação em que o **do-while** pode ser aplicado, pois o valor deve ser lido, calculado e impresso e só

depois deve ser verificado se deve ou não sair do laço. Como não temos o **do-while** no Python, o exercício foi resolvido com **while**, obtendo os mesmos benefícios.

Na resposta proposta, note que a primeira linha, em que foi definido o cabeçalho do **while**, a condição para manter o laço ativo é **1**, como assim? Um número fixo? Isso mesmo, ou seja, esse laço nunca vai acabar, concorda? Pois 1, sempre será 1. Mas, entre as linhas 4 e 7, foi garantido que esse **while** possa terminar caso seja informado um número múltiplo de 6, assim, produzimos um **do-while** com **while**, pois o laço irá executar ao menos uma vez, já que a condição do **while** é sempre verdadeira e a condição que valida se o laço continua ou não, ocorre ao final do laço e após as demais operações terem ocorrido.

3.8 *Loop* infinito na cláusula while

O uso de estruturas de iteração implica em repetir um determinado número de instruções, ou seja, um número finito de repetições, contudo, por erro de programação, é possível provocar um número infinito de repetições, em geral, o nome dado a este efeito é '*loop* infinito' ou 'laço infinito'. Essa situação é indesejável, pois uma vez provocada, o programa não conseguirá concluir a operação e o menor dos problemas será o estouro de memória do computador, ocasionando em travamento. Desta forma, é necessário ter cuidado ao estabelecer a condição de saída do laço. Veja a seguir um exemplo de programa com *loop* infinito.

```
1 resp = 1
2 while resp == 1:
3     idade = int(raw_input("Digite a idade:"))
4     print "A idade e: ", idade
5
6 resp = int(raw_input("Continuar = 1 Terminar = qualquer outro numero"))
```

O exemplo é simples e o erro cometido é mais simples ainda, pois todas as linhas de código necessárias para que o programa funcione estão presentes, contudo a indentação errada de uma linha de código leva este programa há um *loop* infinito. Veja a linha 6, essa linha é responsável por alterar o valor da variável de controle do laço, *resp*, se essa variável é igual a **1**, quer dizer que o usuário deseja continuar digitando as idades, mas se o valor é diferente de **1**, então o programa deve abandonar o laço, mas como dito, no programa acima, o programa nunca irá abandonar o laço, apesar das instruções para mudar o conteúdo da variável estarem lá. Isso ocorre porque a linha 6 foi indentada de forma errada, ela deveria ter sido indentada no bloco de instruções do **while**, pois assim, a cada iteração o usuário seria indagado se o mesmo deseja continuar digitando, como não é o caso, o laço repete, pois *resp* permanece igual a **1** e a linha 6 nunca será executada. A seguir apresento a correção do código.

```
1 resp = 1
2 while resp == 1:
3     idade = int(raw_input("Digite a idade:"))
4     print "A idade e: ", idade
5
6     resp = int(raw_input("Continuar = 1 Terminar = qualquer outro numero"))
```

Veja agora o que mudou, a linha 6 no programa anterior agora pertencem ao bloco de instruções do **while**, note que é um erro simples que pode ocorrer em função de uma indentação errada que não foi notada, mas que poderá causar um bom estrago nos

dados de um programa, dependendo do contexto. Basicamente, os erros que causam *loop* infinito no **while** estão relacionados à variável de controle do laço. Pode ser um esquecimento de atualizar o valor da variável ou mesmo uma comparação com um valor que nunca será atingido na condição do **while**, então basta ficar atento à estes possíveis problemas que os erros serão evitados.

3.9 Exemplos adicionais

É comum utilizar as estruturas de iteração para realizar operações como: contar, somar, calcular a média, obter o mínimo e o máximo. A seguir foram disponibilizados dois exemplos adicionais em forma de exercício, um envolve as operações de contar, somar e calcular a média e o segundo exemplo envolve obter o mínimo e o máximo.

3.9.1 Exercício de Exemplo

Uma transportadora utiliza caminhões que suportam até 10 toneladas de peso, as caixas transportadas tem tamanho fixo e o caminhão comporta no máximo 200 volumes, assim, esta transportadora precisa controlar a quantidade e o peso dos volumes para acomodar nos caminhões. Faça um programa que leia *n* caixas e seu peso, ao final, o programa deve imprimir a quantidade de volumes, o peso total dos volumes e o peso médio dos volumes.

```
1  qtdVolumes = 0
2  pesoTotal = 0
3  pesoMedio = 0
4  resp = int(raw_input("Deseja cadastrar uma caixa? 1-SIM / 2-NAO \n"))
5
6  while resp == 1:
7      qtdVolumes += 1
8      peso = float(raw_input("Informe o peso da caixa:"))
9      pesoTotal += peso
10     resp = int(raw_input("Deseja cadastrar uma caixa? 1-SIM / 2-NAO \n"))
11
12 pesoMedio = pesoTotal / qtdVolumes
13
14 print "Quantidade de volumes: ", qtdVolumes
15 print "Peso total dos volumes: ", pesoTotal
16 print "Peso medio dos volumes: ", pesoMedio
```

O exercício da transportadora pede que seja calculada a quantidade de volumes, calculado o peso total e a média dos pesos, ou seja, é uma situação de contar, somar e calcular a média. Outro detalhe importante é que neste exercício, não sabe-se a quantidade de volumes previamente, então é necessário utilizar o **while**. As linhas 1 a 4 são declarações das variáveis necessárias, note que foi criada uma variável para contar os volumes, *qtdVolumes*, uma variável para somar o peso, *pesoTotal* e uma variável para armazenar o peso médio, *pesoMedio*, essas variáveis estão relacionadas ao ponto chave da resolução do exercício. Entre as linhas 6 e 10 foi adicionado o laço **while** que faz as três operações, lê o peso dos volumes, conta os volumes e soma o peso dos volumes.

Observe que, para contar a quantidade de volumes, foi necessário apenas inicializar a variável *qtdVolumes*, na linha 1 e incrementar a cada iteração na linha 7, pois o número de iterações que o programa irá executar é exatamente igual ao número de volumes. Para somar os pesos, foi igualmente necessário inicializar a variável *pesoTotal* na linha

2 e acumular com a variável *peso* a cada iteração, na linha 9. Veja que para acumular com a variável peso é importante que essa instrução seja posicionada após a leitura do peso. O cálculo da média foi realizado fora do escopo do **while**, na linha 12, pois para calcular a média, primeiro é necessário somar todos os pesos e depois dividir pela quantidade de volumes. Para finalizar, foram incluídas as linhas 14, 15 e 16 que imprimem os resultados desejados.

3.9.2 Exercício de Exemplo

Num frigorífico existem 90 bois. Cada boi traz preso em seu pescoço um cartão contendo seu número de identificação e seu peso. Faça um programa que imprima a identificação e o peso do boi mais gordo e do boi mais magro (supondo que não haja empates).

```
1  boiGordo=0
2  boiMagro=0
3  idBoiGordo = 0
4  idBoiMagro = 0
5
6  for i in range(90):
7      idBoi = int(raw_input("Informe a identificacao do boi: \n"))
8      pesoBoi = float(raw_input("Informe o peso do boi: \n"))
9
10     if pesoBoi > boiGordo:
11         idBoiGordo = idBoi
12         boiGordo = pesoBoi
13
14     if pesoBoi < boiMagro or i == 1:
15         idBoiMagro = idBoi
16         boiMagro = pesoBoi
17
18 print "Identificacao do boi mais gordo: ", idBoiGordo
19 print "Peso do boi mais gordo: ", boiGordo
20 print "Identificacao do boi mais magro: ", idBoiMagro
21 print "Peso do boi mais magro: ", boiMagro
```

No exemplo dos bois, o exercício pede que sejam lidos 90 bois, desta forma, é possível utilizar o **for**, pois sabe-se previamente que são 90, pede também que seja obtido e impresso o boi mais magro e o boi mais gordo, assim, trata-se de um caso de obter o mínimo e o máximo. Além disso, o exercício solicita que seja impresso a identificação e o peso, seja ele do mais magro ou do mais gordo. Note que, declarou-se a variável *idBoi* na linha 7, para ler a identificação do boi e *pesoBoi* na linha 8 para ler o peso do boi, essas duas variáveis é que irão armazenar temporariamente os dados de cada um dos 90 bois. Definiu-se também as variáveis: *idBoiGordo* e *boiGordo* para registrar os dados do boi mais gordo e *idBoiMagro* e *boiMagro* para registrar os dados do boi mais magro, essas são as variáveis chave da resolução do exercício.

A linha 6 do código proposto inicia o laço **for**, entre as linhas 7 e 8 é feita a leitura dos dados do boi da iteração atual. O ponto chave para resolver o exercício está entre as linhas 10 e 16, veja que para obter o boi mais gordo foi necessário inicializar a variável na linha 1 e foi realizada a validação do peso do boi lido com o boi registrado, **pesoBoi > boiGordo**, na linha 10, então analise este cenário! A variável *boiGordo* foi inicializada com zero, assumindo que não exista nenhum boi com peso menor ou igual a zero, assim, é possível concluir que qualquer peso de boi informado será superior a zero, de forma que, na primeira leitura o peso do primeiro boi lido vai substituir o valor da

inicialização nas linhas 11 e 12. Nas iterações seguintes, o peso do novo boi lido só irá substituir o peso e a identificação registrados, caso esse novo peso seja maior que o atual, assim, ao fim das iterações o peso do boi mais gordo será registrado na variável *boiGordo* e a sua respectiva identificação na variável *idBoiGordo*.

O mesmo procedimento é aplicado para o boi magro, foram realizadas apenas duas mudanças, uma é o sinal do operador relacional que passou de **maior (>)** para **menor (<)** e a outra, é que foi adicionado na condição uma disjunção **OU (or)** com a validação **i == 1**. Isso foi feito porque a variável *boiMagro* foi inicializada com zero, pois bem, não há nenhum boi com peso menor que zero, então a condição **i == 1** foi adicionada para garantir que se o laço estiver na primeira iteração ele vai executar as linhas 15 e 16 mesmo que o peso não seja menor, assim, garante-se que o valor inicial zero seja substituído pelo peso do primeiro boi lido, que até o momento será o de menor peso. Em caso de, após isso, ocorre a leitura de um boi com peso inferior, então o peso do novo boi vai ser armazenado na variável *boiMagro* garantindo que os dados do boi mais magro sejam registrados ao fim do laço **for**. Para finalizar o programa, foram incluídas as linhas 18 a 21 que são responsáveis pela impressão dos resultados.

3.10 Resumo da Aula

Nesta aula foram apresentados os conceitos necessários para construir um programa em linguagem Python com estruturas de iteração, ou seja, programas que possibilitem executar diversas tarefas repetidas vezes. Assim, o programa pode por exemplo, possibilitar o cálculo de um reajuste de salário de um número n de funcionários, onde n pode ser previamente conhecido ou não.

Neste sentido, foram apresentadas duas cláusulas de estrutura de iteração em linguagem Python, a cláusula **for** e **while**. A cláusula **for** é muito utilizada para iterar em listas ou executar um número n de operações, quando se conhece previamente este número, é claro. Já a cláusula **while**, em geral, é utilizada quando não é conhecido o número de iterações, assim, essa cláusula permite estabelecer condições de parada que tornam possível a saída de um laço sem conhecer previamente o n.

Não há como dizer que uma cláusula é melhor que a outra, pois, com poucas adaptações, em geral se consegue aplicar o uso das 2 cláusulas em qualquer problema computacional que envolva a repetição de instruções. Contudo, de acordo com o contexto do problema, é possível identificar qual das duas cláusulas se aplica melhor. Desta forma, ao escolher corretamente a cláusula mais apropriada, é provável que o programador irá escrever o código mais eficiente.

3.11 Exercícios da Aula

Os exercícios desta lista foram Adaptados de Lopes e Garcia (2002, p. 136-226).

1. Faça um programa em Python que imprima todos os números de 1 até 100.
2. Faça um programa que imprima todos os números pares de 100 até 1.
3. Faça um programa que imprima os múltiplos de 5, no intervalo de 1 até 500.
4. Faça um programa em Python que permita entrar com o nome, a idade e o sexo de 20 pessoas. O programa deve imprimir o nome da pessoa se ela for do sexo masculino e tiver mais de 21 anos.
5. Sabendo-se que a unidade lógica e aritmética calcula o produto através de somas sucessivas, crie um programa que calcule o produto de dois números inteiros lidos. Suponha que os números lidos sejam positivos e que o multiplicando seja menor do que o multiplicador.
6. Crie um programa em Python que imprima os 20 primeiros termos da série de Fibonacci.
   Observação: os dois primeiros termos desta série são 1 e 1 e os demais são gerados a partir da soma dos anteriores. Exemplo:
   - 1 + 1 = 2, terceiro termo;
   - 1 + 2 = 3, quarto termo, etc.
7. Crie um programa em linguagem Python que permita entrar com o nome, a nota da prova 1 e da prova 2 de 15 alunos. Ao final, imprimir uma listagem, contendo: nome, nota da prova 1, nota da prova 2, e média das notas de cada aluno. Ao final, imprimir a média geral da turma.
8. Faça um programa que permita entrar com o nome e o salário bruto de 10 pessoas. Após ler os dados, imprimir o nome e o valor da alíquota do imposto de renda calculado conforme a tabela a seguir:

Salário	**IRRF**
Salário menor que R$1300,00	Isento
Salário maior ou igual a R$1300,00 e menor que R$2300,00	10% do salário bruto
Salário maior ou igual a R$2300,00	15% do salário bruto

9. No dia da estréia do filme "Procurando Dory", uma grande emissora de TV realizou uma pesquisa logo após o encerramento do filme. Cada espectador respondeu a um questionário no qual constava sua idade e a sua opinião em relação ao filme: excelente - 3; bom - 2; regular - 1. Criar um programa que receba a idade e a opinião de 20 espectadores, calcule e imprima:
   - A média das idades das pessoas que responderam excelente;
   - A quantidade de pessoas que responderam regular;
   - A percentagem de pessoas que responderam bom entre todos os expectadores analisados.
10. Em um campeonato Europeu de Volleyball, se inscreveram 30 países. Sabendo-se que na lista oficial de cada país consta, além de outros dados, peso e idade de 12 jogadores, crie um programa que apresente as seguintes informações:

- O peso médio e a idade média de cada um dos times;
- O atleta mais pesado de cada time;
- O atleta mais jovem de cada time;
- O peso médio e a idade média de todos os participantes.

11. Construa um programa em Python que leia vários números e informe quantos números entre 100 e 200 foram digitados. Quando o valor 0 (zero) for lido, o algoritmo deverá cessar sua execução.

12. Dado um país A, com 5 milhões de habitantes e uma taxa de natalidade de 3% ao ano, e um país B com 7 milhões de habitantes e uma taxa de natalidade de 2% ao ano, fazer um programa que calcule e imprima o tempo necessário para que a população do país A ultrapasse a população do país B.

13. Uma empresa de fornecimento de energia elétrica faz a leitura mensal dos medidores de consumo. Para cada consumidor, são digitados os seguintes dados:

    - número do consumidor
    - quantidade de kWh consumidos durante o mês
    - tipo (código) do consumidor
      1-residencial, preço em reais por kWh = 0,3
      2-comercial, preço em reais por kWh = 0,5
      3-industrial, preço em reais por kWh = 0,7

    Os dados devem ser lidos até que seja encontrado o consumidor com número 0 (zero). O programa deve calcular e imprimir:

    - O custo total para cada consumidor
    - O total de consumo para os três tipos de consumidor
    - A média de consumo dos tipos 1 e 2

14. Faça um programa que leia vários números inteiros e apresente o fatorial de cada número. O algoritmo encerra quando se digita um número menor do que 1.

15. Faça um programa em Python que permita entrar com a idade de várias pessoas e imprima:

    - total de pessoas com menos de 21 anos
    - total de pessoas com mais de 50 anos

16. Sabendo-se que a unidade lógica e aritmética calcula a divisão por meio de subtrações sucessivas, criar um algoritmo que calcule e imprima o resto da divisão de números inteiros lidos. Para isso, basta subtrair o divisor ao dividendo, sucessivamente, até que o resultado seja menor do que o divisor. O número de subtrações realizadas corresponde ao quociente inteiro e o valor restante da subtração corresponde ao resto. Suponha que os números lidos sejam positivos e que o dividendo seja maior do que o divisor.

17. Crie um programa em Python que possa ler um conjunto de pedidos de compra e calcule o valor total da compra. Cada pedido é composto pelos seguintes campos:

    - número de pedido

- data do pedido (dia, mês, ano)
- preço unitário
- quantidade

O programa deverá processar novos pedidos até que o usuário digite 0 (zero) como número do pedido.

18. Uma pousada estipulou o preço para a diária em R$30,00 e mais uma taxa de serviços diários de:

    - R$15,00, se o número de dias for menor que 10;
    - R$8,00, se o número de dias for maior ou igual a 10;

    Faça um programa que imprima o nome, a conta e o número da conta de cada cliente e ao final o total faturado pela pousada.

    O programa deverá ler novos clientes até que o usuário digite 0 (zero) como número da conta.

19. Em uma Universidade, os alunos das turmas de informática fizeram uma prova de algoritmos. Cada turma possui um número de alunos. Criar um programa que imprima:

    - quantidade de alunos aprovados;
    - média de cada turma;
    - percentual de reprovados.

    Obs.: Considere aprovado com nota >= 7.0

20. Uma pesquisa de opinião realizada no Rio de Janeiro, teve as seguintes perguntas:

    - Qual o seu time de coração?
      1-Fluminense;
      2-Botafogo;
      3-Vasco;
      4-Flamengo;
      5-Outros
    - Onde você mora?
      1-RJ;
      2-Niterói;
      3-Outros
    - Qual o seu salário?

    Faça um programa que imprima:

    - o número de torcedores por clube;
    - a média salarial dos torcedores do Botafogo;
    - o número de pessoas moradoras do Rio de Janeiro, torcedores de outros clubes;
    - o número de pessoas de Niterói torcedoras do Fluminense

Obs.: O programa encerra quando se digita 0 para o time.

21. Em uma universidade cada aluno possui os seguintes dados:

    - Renda pessoal;
    - Renda familiar;
    - Total gasto com alimentação;
    - Total gasto com outras despesas;

    Faça um programa que imprima a porcentagem dos alunos que gasta acima de R$200,00 com outras despesas. O número de alunos com renda pessoal maior que a renda familiar e a porcentagem gasta com alimentação e outras despesas em relação às rendas pessoal e familiar.

    Obs.: O programa encerra quando se digita 0 para a renda pessoal.

22. Crie um programa que ajude o DETRAN a saber o total de recursos que foram arrecadados com a aplicação de multas de trânsito.

    O algoritmo deve ler as seguintes informações para cada motorista:

    - número da carteira de motorista (de 1 a 4327);
    - número de multas;
    - valor de cada uma das multas.

    Deve ser impresso o valor da dívida para cada motorista e ao final da leitura o total de recursos arrecadados (somatório de todas as multas). O programa deverá imprimir também o número da carteira do motorista que obteve o maior número de multas.

    Obs.: O programa encerra ao ler a carteira de motorista de valor 0.

23. Crie um programa que leia um conjunto de informações (nome, sexo, idade, peso e altura) dos atletas que participaram de uma olimpíada, e informar:

    - a atleta do sexo feminino mais alta;
    - o atleta do sexo masculino mais pesado;
    - a média de idade dos atletas.

    Obs.: Deverão se lidos dados dos atletas até que seja digitado o nome @ para um atleta.

24. Faça um programa que calcule quantos litros de gasolina são usados em uma viagem, sabendo que um carro faz 10 km/litro. O usuário fornecerá a velocidade do carro e o período de tempo que viaja nesta velocidade para cada trecho do percurso. Então, usando as fórmulas: $distancia = tempo \times velocidade$ e $litrosconsumidos = \frac{distancia}{10}$, o programa computará, para todos os valores não-negativos de velocidade, os litros de combustível consumidos. O programa deverá imprimir a distância e o número de litros de combustível gastos naquele trecho. Deverá imprimir também o total de litros gastos na viagem. O programa encerra quando o usuário informar um valor negativo de velocidade.

25. Faça um programa que calcule o imposto de renda de um grupo de contribuintes, considerando que:

a) os dados de cada contribuinte (CIC, número de dependentes e renda bruta anual) serão fornecidos pelo usuário via teclado;

b) para cada contribuinte será feito um abatimento de R$600 por dependente;

c) a renda líquida é obtida diminuindo-se o abatimento com os dependentes da renda bruta anual;

d) para saber quanto o contribuinte deve pagar de imposto, utiliza-se a tabela a seguir:

Renda Líquida	Imposto
até R$1000	Isento
de R$1001 a R$5000	15%
acima de R$5000	25%

e) o valor de CIC igual a zero indica final de dados;

f) o programa deverá imprimir, para cada contribuinte, o número do CIC e o imposto a ser pago;

g) ao final o programa deverá imprimir o total do imposto arrecadado pela Receita Federal e o número de contribuintes isentos;

h) leve em consideração o fato de o primeiro CIC informado poder ser zero.

26. Foi feita uma pesquisa de audiência de canal de TV em várias casas de uma certa cidade, em um determinado dia. Para cada casa visitada foram fornecidos o número do canal (4, 5, 7, 12) e o número de pessoas que estavam assistindo a ele naquela casa. Se a televisão estivesse desligada, nada seria anotado, ou seja, esta casa não entraria na pesquisa. Criar um programa que:

    - Leia um número indeterminado de dados, isto é, o número do canal e o número de pessoas que estavam assistindo;
    - Calcule e imprima a porcentagem de audiência em cada canal.

    Obs.: Para encerrar a entrada de dados, digite o número do canal zero.

27. Crie um programa que calcule e imprima o CR do período para os alunos de computação. Para cada aluno, o algoritmo deverá ler:

    - número da matrícula;
    - quantidade de disciplinas cursadas;
    - notas em cada disciplina;
      Além do CR de cada aluno, o programa deve imprimir o melhor CR dos alunos que cursaram 5 ou mais disciplinas.
    - fim da entrada de dados é marcada por uma matrícula inválida (matrículas válidas de 1 a 5000);
    - CR do aluno é igual à média aritmética de suas notas.

28. Construa um programa que receba a idade, a altura e o peso de várias pessoas, Calcule e imprima:

    - a quantidade de pessoas com idade superior a 50 anos;
    - a média das alturas das pessoas com idade entre 10 e 20 anos;

- a porcentagem de pessoas com peso inferior a 40 quilos entre todas as pessoas analisadas.

29. Construa um programa que receba o valor e o código de várias mercadorias vendidas em um determinado dia. Os códigos obedecem a lista a seguir:

    L-limpeza
    A-Alimentação
    H-Higiene
    Calcule e imprima:

    - o total vendido naquele dia, com todos os códigos juntos;
    - o total vendido naquele dia em cada um dos códigos.

    Obs.: Para encerrar a entrada de dados, digite o valor da mercadoria zero.

30. Faça um programa que receba a idade e o estado civil (C-casado, S-solteiro, V-viúvo e D-desquitado ou separado) de várias pessoas. Calcule e imprima:

    - a quantidade de pessoas casadas;
    - a quantidade de pessoas solteiras;
    - a média das idades das pessoas viúvas;
    - a porcentagem de pessoas desquitadas ou separadas dentre todas as pessoas analisadas.

    Obs.: Para encerrar a entrada de dados, digite um número menor que zero para a idade.

AULA 4

Listas

Metas da Aula

1. Entender e praticar os conceitos do uso de listas na linguagem Python.
2. Aplicar variadas situações relacionadas ao uso de listas em programação.
3. Escrever programas que farão uso de listas.

Ao término desta aula, você será capaz de:

1. Declarar uma lista.
2. Obter os valores ou elementos de uma lista.
3. Atribuir valores ou elementos em uma lista.
4. Utilizar estrutura de iteração com listas.

4.1 Listas

Até o momento, foram utilizadas variáveis em que é possível armazenar 1 (um) dado, assim, se for declarada uma variável, o programa poderá armazenar um dado apenas, na memória associada a esta variável. Desta forma, se for necessário armazenar, por exemplo, a idade de 2 pessoas, duas variáveis devem ser declaradas, se for necessário armazenar a idade de 3 pessoas, então 3 variáveis devem ser definidas, mas e se for necessário armazenar a idade de 10 ou 100 pessoas? Não seria nada prático declarar 10 variáveis, muito menos declarar 100, até mesmo porque o maior esforço seria no controle dessas diversas variáveis. Neste caso é útil o uso de listas.

As listas são capazes de armazenar várias informações, inclusive são de capazes de armazenar objetos e até outras listas, de fato elas são bem flexíveis, podendo aumentar e diminuir conforme a necessidade (LUTZ; ASCHER, 2007, p. 117). Ao declarar uma lista, não é preciso informar o seu tamanho, pois, como já mencionado, as listas podem aumentar ou diminuir de tamanho facilmente, então basta declarar a lista como vazia e depois manipular. Veja a seguir a sintaxe para a declaração de uma lista.

```
1 nomeDaLista = []
```

Conforme podemos ver na sintaxe, é muito fácil, basta definir o nome e atribuir os colchetes para que ela seja considerada uma lista vazia. Naturalmente, uma lista pode ser preenchida no momento da declaração, caso o programador já disponha dos valores. Veja a seguir alguns exemplos:

```
1 >>> idades = [10, 16, 14, 18, 5]
2 >>> notas = [5.5, 7.8, 9,2, 1.3, 5,7]
3 >>> precos = [20.7, 50.4, 30.2]
4 >>> tabela = [10, 20, 15.5, [6, 5, 9]]
```

Na linha 1 do exemplo, foi declarada uma lista com valores inteiros cujo nome é *idade*. Como pode-se observar, a atribuição no momento da declaração é feita incluindo os valores entre chaves e separados por vírgula. Apesar de incluir os valores iniciais na lista, não há impedimento para ajustar os valores, diminuir ou aumentar a quantidade de elementos da lista. As declarações das linhas 2 e 3 seguem o mesmo padrão das anteriores, o que muda é só o fato de que nestes casos os valores iniciais são pertencentes ao conjunto dos reais. O exemplo de declaração da linha 4, foi adicionado apenas para exemplificar que é possível inserir elementos em uma lista com formatos distintos, inclusive na dimensão e tipo. Note que, os dois primeiros elementos são do tipo inteiro, o terceiro elemento é do tipo real e o quarto elemento é uma outra lista que possui três elementos do tipo inteiro.

4.2 Operações básicas em uma lista

Há várias operações simples que podem ser realizadas com listas e que irão facilitar o trabalho do programador, como verificar o tamanho da lista, concatenar duas listas, repetir valores em lista e acessar os valores da lista. Veja a seguir alguns exemplos dessas operações e a explicação na sequência.

```
1 >>> idades = [10, 16, 14, 18, 5]
2 >>> len(idades)
3 5
```

Começando pela verificação do tamanho da lista, é muito simples como apresentado no exemplo. Note que, na linha 1 a lista foi declarada e preenchida com 5 valores. Na linha 2 a função **len()** foi acionada para obter o tamanho da lista que, foi exibido na linha 3. Assim, para obter o tamanho da lista, basta utilizar a função **len()** passando a lista como argumento da função. Vejamos agora outras operações básicas no próximo exemplo.

```
1 >>> idades1 = [10, 16, 14, 18, 5]
2 >>> idades2 = [5, 10, 35]
3 >>> idades = idades1 + idades2
4 >>> idades
5 [10, 16, 14, 18, 5, 5, 10, 35]
6 >>> [30] * 5
7 [30, 30, 30, 30, 30]
8 >>> 18 in idades
9 True
```

Observe que nas linhas 1 e 2 foram declarados duas listas, sendo *idades1* com valores inteiros e *idades2*, também com valores inteiros, até ai, nenhuma novidade. Na linha 3 foi realizada uma soma das duas listas anteriores e o resultado da soma foi atribuído a outra lista, cujo nome é *idades*. Essa soma, não irá realizar uma soma propriamente dita, mas uma concatenação das listas, isso pode ser verificado na linha 5, note que ao exibir o conteúdo da lista *idades*, o resultado é a concatenação das outras duas listas. Na linha 6 vemos um exemplo de repetição de valores em lista, veja que ao declarar o primeiro elemento e multiplicar pelo número de repetições que se deseja, foi obtida uma lista com valores repetidos, conforme a linha 7. Por fim, na linha 8 temos um exemplo de avaliação de conteúdo da lista, pois neste caso, está sendo avaliado se o elemento 18 está contido na lista *idades*, como ele está contido, então o interpretador retornou **True** na linha 9. Vejamos agora exemplos de como acessar os elementos da lista.

```
1  >>> idades
2  [10, 16, 14, 18, 5, 5, 10, 35]
3  >>> idades[0]
4  10
5  >>> idades[3]
6  18
7  >>> idades[9]
8  Traceback (most recent call last):
9
10   File "<ipython-input-18-59ba8a2ef0d2>", line 1, in <module>
11     idades[9]
12
13 IndexError: list index out of range
```

No código de exemplo, temos a forma direta de acessar o elemento de uma lista, pelo índice da posição do elemento na lista. Veja que na linha 3, obteve-se o valor 10, relativo à posição 0 da lista *idades*, mas qual é a posição 0? É a primeira posição, sim, a indexação dos elementos da lista começa na posição 0. Naturalmente, se inicia com 0, então termina sempre com *N-1*, sendo *N* o número de elementos da lista. Na linha 5, foi acessado o valor 18 da posição 3, que na verdade refere-se ao quarto elemento da lista, por fim, na linha 7, tentou-se acessar o elemento da posição 9, contudo o interpretador não foi capaz de acessar, pois o elemento não existe na lista, assim, retornou uma mensagem de erro, conforme pode ser visto nas linhas seguintes. As listas permitem o seu fracionamento, veja seguir alguns exemplos:

```
1  >>> idades
2  [10, 16, 14, 18, 5, 5, 10, 35]
3  >>> idades[-2]
4  10
5  >>> idades[3:]
6  [18, 5, 5, 10, 35]
7  >>> idades[:4]
8  [10, 16, 14, 18]
9  >>> idades[3:6]
10 [18, 5, 5]
```

Foram apresentadas algumas das possibilidades de fracionamento neste exemplo, contudo há várias outras possibilidades que podem ser exploradas conforme a sua necessidade. Na linha 3 foi acessada a lista passando o índice com valor negativo, quando isso é feito a contagem das posições ocorrerá da direita para a esquerda, que no caso será o penúltimo elemento. Na linha 5, ao colocar o sinal de **:** após o índice 3, é como se solicitássemos à lista: me dê todos os elementos a partir do índice 3, inclusive. O contrário ocorre quando colocamos o sinal de **:** antes do índice, que é caso da linha 7, em que foram solicitados os elementos do primeiro até o elemento de índice 4, exclusive. Já na linha 9, foi solicitado à lista os valores entre o índice 3, inclusive e o índice 6, exclusive. Outra forma de acessar os valores é obtendo os elementos da lista em uma iteração de um laço, veja a seguir o exemplo.

```
1  >>> for x in idades:
2          print x,
3
4  10 16 14 18 5 5 10 35
```

A cláusula **for** estudada na aula 3 é bem útil neste caso, veja que, atribuímos um "apelido", no caso, *x*, para os elementos que serão acessados, assim, a cada iteração do laço na lista, o elemento atual que está sendo visitado é o *x*, de forma que, podemos acessá-lo por completo, e fazer o uso necessário, no caso do exemplo, o *x* é impresso a cada iteração.

4.3 Matrizes baseadas em listas

Como é possível adicionar outras listas em uma lista, então, a forma mais simples de representar matrizes em Python é utilizando listas com sub-listas (LUTZ; ASCHER, 2007, p. 120). Veja a seguir um exemplo:

```
1  >>> matNotas = [[7, 8.3, 9.2], [3, 5.5, 6.2], [8, 7.2, 7.7]]
2  >>> matNotas[0]
3  [7, 8.3, 9.2]
4  >>> matNotas[0][1]
5  8.3
```

Na primeira linha temos a declaração de uma matriz baseada em lista com sub-listas de dimensão **3 x 3**, cujo nome é *matNotas*. Na linha 2 foi solicitado o acesso do elemento da primeira posição da lista, como cada elemento da lista é uma sub-lista, então foi exibida a sub-lista inteira, como pode ser visto na linha 3. Mas, como acessar um elemento da sub-lista? É fácil, basta indicar a posição da sub-lista conforme o exemplo na linha 4, note que foi realizado o acesso do primeiro elemento da lista, de índice 0, e o segundo elemento da sub-lista, de índice 1.

4.4 Alterando as listas

Trataremos inicialmente da alteração no local, ou seja, modificando o elemento contido na lista diretamente. O conteúdo pode ser atribuído a um item ou a uma faixa de itens. Veja a seguir mais um trecho de código-fonte com exemplos de atribuição.

```
1  >>> idades = [10, 16, 14, 18, 5]
2  >>> idades[2] = 12
3  >>> idades
4  [10, 16, 12, 18, 5]
5  >>> idades[0:2] = [13, 11]
6  >>> idades
7  [13, 11, 12, 18, 5]
```

Na linha 1 do código de exemplo foi declarada a lista *idades* com 5 valores inicias. Na linha 2 foi atribuído o valor **12** na posição 3, índice 2, da lista, substituindo o valor novo pelo antigo, como pode ser visto na linha 4. Na linha 5 foram alterados 2 valores da faixa entre a posição 1 e a posição 2, índices 1 até 2, exclusive. O resultado final é apresentado na linha 7. Neste caso, é importante notar que os elementos após a faixa delimitada são mantidos.

4.5 Listas são objetos mutáveis

As listas em Python são objetos mutáveis, isso quer dizer que não é possível fazer uma cópia de uma lista da mesma forma que 'copiamos' variáveis. A verdade é que também não copiamos variáveis, mas quando atribuímos uma variável a outra e depois alteramos um valor em uma delas, essa segunda variável passa a referenciar o novo valor, e outra variável continua referenciando o valor antigo, o que dá a impressão de que foi feita uma cópia da variável. Veja a seguir o exemplo deste efeito.

```
1  >>> x = 10
2  >>> y = x
3  >>> x
4  10
5  >>> y
6  10
7  >>> x = 20
8  >>> x
9  20
10 >>> y
11 10
```

Observe que foi atribuído um valor à x na primeira linha, e depois foi declarada a variável y, sendo que y recebeu x, mas tecnicamente falando, y na verdade passou a referenciar o mesmo objeto, o valor 10, que x referenciava. Por isso, ao visualizar o valor de x na linha 3 e de y na linha 5, o valor impresso é o mesmo nas linhas 4 e 6. Contudo, ao atribuir o valor 20 à x na linha 7, x deixou de referenciar o valor 10 e passou a referenciar o valor 20, ao passo que y continua referenciando o valor 10, como pode-se observar nas linhas 10 e 11. Como as listas são mutáveis, não dá para fazer a mesma coisa, pois ao atribuir um novo valor à lista, ela vai mudar, já que é mutável, sem perder os valores anteriores, assim, os dois nomes que referenciam a lista vão continuar referenciando o mesmo local. Veja a seguir o exemplo.

```
1  >>> lista1 = [1, 2, 3]
2  >>> lista2 = lista1
3  >>> lista1
4  [1, 2, 3]
5  >>> lista2
6  [1, 2, 3]
7  >>> lista1[0] = 2
8  >>> lista1
9  [2, 2, 3]
10 >>> lista2
11 [2, 2, 3]
```

Observe que na linha 1 foi declarada *lista1* e na linha 2, foi declarada *lista2* recebendo *lista1*, da mesma forma que foi realizado com as variáveis no exemplo anterior e da mesma forma, *lista2* passou a referenciar o mesmo objeto, no caso a lista, que é referenciado por *lista1*. Por isso, os valores são iguais ao imprimir, nas linhas 3 a 6. Ao mudar o valor de uma posição da *lista1*, note que a *lista2* também sofreu alteração, conforme as linhas 10 e 11, mas, na prática, *lista2* não mudou, ocorre que ela apenas faz referência para a mesma lista referenciada por *lista1*, e como a lista é mutável, ao alterar o conteúdo da lista, não foi criado um novo objeto, como ocorreu com as variáveis. Mas, e se precisarmos fazer uma cópia da lista, de forma que, ao alterar uma, a outra não se altere? Existem várias formas de fazer isso, dentre elas, há uma bem simples, conforme o exemplo a seguir:

```
1  >>> import copy
2  >>> lista1 = [1, 2, 3]
3  >>> lista2 = copy.copy(lista1)
4  >>> lista1
5  [1, 2, 3]
6  >>> lista2
7  [1, 2, 3]
8  >>> lista1[0] = 2
9  >>> lista1
10 [2, 2, 3]
11 >>> lista2
12 [1, 2, 3]
```

Na linha 1 foi realizado um **import** para a biblioteca **copy**, essa biblioteca vai permitir que façamos uma cópia da lista, ao invés de criar uma nova referência. Na linha 2 foi declarada a *lista1* igual ao exemplo anterior, na linha 3 foi declarada a *lista2*, que recebeu a copia da *lista1*, observe que foi utilizada a função **copy**() passando *lista1* como argumento. Para mostrar que foi feita a cópia, as duas listas foram impressas nas linhas seguintes e na linha 8 foi alterado o valor de uma posição da *lista1*, agora veja que nas linhas 11 e 12, os valores referenciados pela *lista2* permanecem inalterados mesmo após alterar um valor referenciado pela *lista1*, ou seja, as duas listas estão referenciando áreas de memória distintas.

4.6 Métodos da lista

As listas em Python possuem métodos prontos que desempenham funções muito úteis, como adicionar um elemento, ordenar a listas, etc. Veja a seguir alguns exemplos:

```
1  >>> idades
2  [13, 11, 12, 18, 5]
3  >>> idades.append(8)
4  >>> idades
5  [13, 11, 12, 18, 5, 8]
6  >>> idades.sort()
7  >>> idades
8  [5, 8, 11, 12, 13, 18]
9  >>> idades.sort(reverse=True)
10 >>> idades
11 [18, 13, 12, 11, 8, 5]
```

Na linha 3, veja que foi invocado um método, **append()**, que pertence à lista *idades*. Na verdade, qualquer lista declarada pelo programador irá disponibilizar vários métodos, dentre eles, os aqui apresentados. O método **append()** adiciona um novo elemento ao final da lista, como pode se verificar na linha 5. Na linha 6, foi invocado outro método, o **sort()**, este ordena os elementos da lista, conforme o resultado na linha 8. O **sort()** permite também ordenar uma lista de forma descendente, conforme pode a linha 9. Em relação ao **sort()**, é importante ressaltar que ao acionar o método, conforme a linha 6, a lista *idades* foi originalmente alterada, mas e se o programador precisar manter a lista original desordenada? Então neste caso, poderá utilizar o método **sorted()** conforme o exemplo a seguir.

```
1  >>> idades = [13, 11, 12, 18, 5, 8]
2  >>> novaIdades = sorted(idades)
3  >>> idades
4  [13, 11, 12, 18, 5, 8]
5  >>> novaIdades
6  [5, 8, 11, 12, 13, 18]
7  >>> novaIdades = sorted(idades, reverse=True)
8  [18, 13, 12, 11, 8, 5]
9  >>> idades
10 [13, 11, 12, 18, 5, 8]
```

Na linha 1 foi realizada a declaração da lista *idades* com os valores iniciais desordenados. Na linha 2 foi feita a invocação da função **sorted()** passando como argumento a lista *idades*. Observe que na linha 3, ao exibir os dados da lista *idades*, ela permaneceu com os dados inalterados, na linha 4, ao passo na que linha 5, ao exibir os dados da lista *novaIdades*, os dados são exibidos em ordem crescente, isso ocorreu porque a função **sorted()** retorna uma nova lista ordenada com base na lista que recebeu como argumento, sem alterar a lista original. A função **sorted()** também permite ordenar em ordem decrescente, conforme pode ser visto nas linhas 7 e 8.

Pode ocorrer de necessitarmos obter os índices da posição original de uma lista, mas ordenados. Como assim? Bem, imagine que você tivesse duas informações na lista original, o dado e o índice, de forma que, ao ordenar pelo dado, o índice fosse realocado conforme a nova posição relativa de seu dado. Entendeu? Isso pode ser muito útil em alguns casos, como na implementação da heurística gulosa. A biblioteca **NumPy** disponibiliza uma função que retorna exatamente isso. Veja a seguir o exemplo de uso da função **argsort()**.

```
1  >>> import numpy as np
2  >>> idades
3  >>> [13, 11, 12, 18, 5, 8]
```

```
4  >>> indIdades = np.argsort(idades)
5  >>> idades
6  >>> [13, 11, 12, 18, 5, 8]
7  >>> indIdades
8  >>> array([4, 5, 1, 2, 0, 3], dtype=int64)
9  >>> indIdades = np.argsort(idades)[::-1]
10 >>> indIdades
11 >>> array([3, 0, 2, 1, 5, 4], dtype=int64)
```

Como pode ver na linha 1 do exemplo, é necessário importar a biblioteca **NumPy** para fazer uso de suas funções. Na linha 2 foi exibido os dados da lista *idades*, apenas para confirmar que continuam desordenados. Na linha 4 foi invocada a função **argsort()** da biblioteca **NumPy**, observe que a invocação é precedida pelo apelido que foi dado à biblioteca ao importá-la. Como a intenção é apenas ordenar em ordem crescente, bastou passar a lista *idades*, como argumento e a função retornou uma nova lista com base na lista original, salvando o resultado na lista *indIdades*.

Observe na linha 8, que ao exibir os dados da lista *indIdades* são exibidas as informações do índice relativo após ordenação, exemplo, o primeiro elemento é o índice **4** que equivale ao valor **5** da lista original, que por sua vez é o menor valor, ou seja, após ordenar a função **argsort()**, ao invés de retornar a lista ordenada, ela retorna a lista dos índices dos elementos após ordenados, com base na lista original. Veja nas linhas 9 à 10 que é possível utilizar **argsort()** em ordem inversa também, basta ao final da invocação da função, incluir os parâmetros: **[::-1]**. Veja a seguir outros métodos que podem ser úteis.

```
1  >>> idades = [13, 11, 12, 18, 5, 8]
2  >>> idades.remove(12)
3  >>> idades
4  [13, 11, 18, 5, 8]
5  >>> idades.pop()
6  8
7  >>> idades
8  [13, 11, 18, 5]
```

Na linha 2 foi acionado o método **remove()**, note que ele remove o elemento da lista pelo seu valor e não pelo índice, o resultado após a remoção é apresentado na linha 4. Outro método que remove um elemento da lista é o **pop()**, do exemplo da linha 5, contudo, **pop()** remove sempre no final da lista, por isso, ele não recebe um valor como argumento. Na lista de exemplo, o último elemento é o **8**, então ele foi removido, como pode ser visto na linha 6. Vejamos agora um exercício de exemplo.

4.6.1 Exercício de Exemplo

Faça um programa em Python que armazene 15 números inteiros em uma lista NUM e imprima uma listagem dos números lidos.
Fonte: Adaptado de Lopes e Garcia (2002, p. 278)

```
1  NUM = []
2  #lendo os valores
3  for i in range(15):
4      numero = int(raw_input("Informe um numero:\n"))
5      NUM.append(numero)
6
7  #imprimindo os valores
```

```
8 for x in NUM:
9     print "Numero: ", x
```

O código-fonte proposto como resposta do exercício mostra como é possível utilizar uma estrutura de iteração, no caso o **for**, para facilitar o trabalho ao utilizar listas. Veja que na linha 1 foi declarada a lista vazia *NUM*, para atender ao solicitado no enunciado do exercício. Assim, a lista *NUM* vai armazenar os 15 valores inteiros lidos. Para ler facilmente os valores, foi utilizado um **for** na linha 3, note que, como a nossa lista está vazia e o número de elementos a serem inseridos é fixo, 15, então utilizou-se o **range**() no **for**. Entre as linhas 8 e 9, há um segundo **for**, pois neste caso, o objetivo é após ler todos os valores, nas linhas 3 a 5, imprimir todos eles, o que é realizado na linha 9 pelo **print**.

4.7 Resumo da Aula

Esta aula trouxe conceitos importantes sobre o uso de listas, que são uma estrutura heterogênea que permite o armazenamento de *n* valores com fácil acesso por meio de índices ou iterando pelos elementos. Assim, foram apresentados conceitos e exemplos práticos na linguagem Python que permitem implementar operações de atribuição de valores na lista, o acesso aos valores e o uso de estruturas de iteração para facilitar a sua manipulação.

Para atribuir e acessar valores em uma lista, basta utilizar o índice da posição do elemento, que inicia com **0 (zero)** e termina com *n-1*, sendo *n* o tamanho da lista. Cada acesso, seja atribuição ou leitura de um valor requer uma instrução do programa, o que poderia levar o programador a ter de escrever centenas de instruções, em alguns casos milhares, mas, felizmente as estruturas de iteração permitem eliminar essa necessidade tornando possível o acesso as listas com milhares de posições em algumas poucas linhas de código.

Para finalizar foram apresentadas algumas funções que podem ser utilizadas em conjunto com as listas, como: **append()**, **sort()**, **sorted()**, **argsort()**, **remove()** e **pop()**. É importante salientar que há dezenas de outras funções disponíveis que podem facilitar o desenvolvimento de operações em listas, uma boa busca na internet poderá revelar muitas outras funções.

4.8 Exercícios da Aula

Os exercícios desta lista foram Adaptados de Lopes e Garcia (2002, p. 277-303).

1. Faça um programa em Python que armazene 15 números inteiros em uma lista e depois permita que o usuário digite um número inteiro para ser buscado na lista, se for encontrado o programa deve imprimir a posição desse número na lista, caso contrário, deve imprimir a mensagem: "Nao encontrado!".

2. Faça um programa que armazene 10 letras em uma lista e imprima uma listagem numerada.

3. Construa uma programa em Python que armazene 15 números em uma lista e imprima uma listagem numerada contendo o número e uma das mensagens: par ou ímpar.

4. Faça um programa que armazene 8 números em uma lista e imprima todos os números. Ao final, imprima o total de números múltiplos de seis.

5. Faça um programa que armazene as notas das provas 1 e 2 de 15 alunos. Calcule e armazene a média arredondada. Armazene também a situação do aluno: 1-Aprovado ou 2-Reprovado. Ao final o programa deve imprimir uma listagem contendo as notas, a média e a situação de cada aluno em formato tabulado. Utilize quantas listas forem necessários para armazenar os dados.

6. Construa um programa que permita armazenar o salário de 20 pessoas. Calcular e armazenar o novo salário sabendo-se que o reajuste foi de 8%. Imprimir uma listagem numerada com o salário e o novo salário. Declare quantas listas forem necessários.

7. Crie um programa que leia o preço de compra e o preço de venda de 100 mercadorias (utilize listas). Ao final, o programa deverá imprimir quantas mercadorias proporcionam:

   - lucro < 10%
   - 10% <= lucro <= 20%
   - lucro > 20%

8. Construa um programa que armazene o código, a quantidade, o valor de compra e o valor de venda de 30 produtos. A listagem pode ser de todos os produtos ou somente de um ao se digitar o código.

9. Faça um programa em Python que leia dois conjuntos de números inteiros, tendo cada um 10 elementos. Ao final o programa deve listar os elementos comuns aos conjuntos.

10. Faça um programa que leia uma lista *listaL* de 10 elementos e obtenha uma lista *listaW* cujos componentes são os fatoriais dos respectivos componentes de *listaL*.

11. Construa um programa que leia dados para uma lista de 100 elementos inteiros. Imprimir o maior e o menor, sem ordenar, o percentual de números pares e a média dos elementos da lista.

12. Crie um programa para gerenciar um sistema de reservas de mesas em uma casa de espetáculo. A casa possui 30 mesas de 5 lugares cada. O programa deverá permitir que o usuário escolha o código de uma mesa (100 a 129) e forneça a quantidade de lugares desejados. O programa deverá informar se foi possível realizar a reserva e atualizar a reserva. Se não for possível, o programa deverá emitir uma mensagem. O programa deve terminar quando o usuário digitar o código **0 (zero)** para uma mesa ou quando todos os 150 lugares estiverem ocupados.

13. Construa um programa que realize as reservas de passagens áreas de uma companhia. O programa deve permitir cadastrar o número de 10 voos e definir a quantidade de lugares disponíveis para cada um. Após o cadastro, leia vários pedidos de reserva, constituídos do número da carteira de identidade do cliente e do número do voo desejado. Para cada cliente, verificar se há possibilidade no voo desejado. Em caso afirmativo, imprimir o número da identidade do cliente e o número do voo, atualizando o número de lugares disponíveis. Caso contrário, avisar ao cliente a inexistência de lugares. A leitura do número 0 (zero) para o voo desejado indica o término da leitura de reservas.

14. Faça um programa que armazene 50 números inteiros em uma lista. O programa deve gerar e imprimir uma segunda lista em que cada elemento é o quadrado do elemento da primeira lista.

15. Faça um programa que leia e armazene vários números, até digitar o número 0. Imprimir quantos números iguais ao último número foram lidos. O limite de números é 100.

16. Crie um programa em Python para ler um conjunto de 100 números reais e informe:

    - quantos números lidos são iguais a 30
    - quantos são maior que a média
    - quantos são iguais a média

17. Faça um programa que leia um conjunto de 30 valores inteiros, armazene-os em uma lista e os imprima ao contrário da ordem de leitura.

18. Faça um programa em Python que permita entrar com dados para uma lista *L*, em que podem existir vários elementos repetidos. Gere uma lista *L2* ordenada a partir da lista *L* e que terá apenas os elementos não repetidos.

19. Suponha duas listas de 30 elementos cada, contendo: código e telefone. Faça um programa que permita buscar pelo código e imprimir o telefone.

20. Faça um programa que leia a matrícula e a média de 100 alunos. Ordene da maior para a menor nota e imprima uma relação contendo todas as matrículas e médias.

AULA 5

Funções

Metas da Aula

1. Entender e praticar os conceitos do uso de funções na linguagem Python.
2. Aplicar variadas situações relacionadas ao uso de funções em programação.
3. Aprender a escrever novas funções em linguagem Python.

Ao término desta aula, você será capaz de:

1. Definir uma função.
2. Escrever novas funções em linguagem Python.

5.1 Funções

As funções estão presentes em todo o código de um programa, embora nas aulas anteriores elas não tenham sido mencionadas, vários programas construidos fizeram uso de funções. Exemplos de funções utilizadas, são: **raw_input()**, **print**, **len()**, entre outras. As funções são blocos de código encapsulado e que podem ser reutilizados ao longo do programa (LUTZ; ASCHER, 2007; CRUZ, 2017), assim, a função **print**, por exemplo, trata-se de um bloco de instruções que é capaz de produzir a saída de uma informação para o usuário do programa. Como, em vários momentos do ciclo de vida do programa, se faz necessário apresentar uma informação por um dispositivo de saída, então é conveniente que as instruções responsáveis por esta ação, sejam encapsuladas em uma função, para que a ação possa ser invocada a qualquer momento no programa.

Desta forma, podemos entender que as funções são úteis quando identificamos trechos de código que, em geral, serão reutilizados em várias partes do programa. Assim, o programador pode criar as suas próprias funções para reutilizar o código-fonte que julgar necessário. Além disso, em geral, a linguagem Python, está acompanhada de bibliotecas que trazem outras várias funções que facilitam a vida do programador, adicionalmente, há várias bibliotecas extras que podem ser instaladas em conjunto com o Python, como: **NumPy**, **SciPy**, **NLTK** (MCKINNEY, 2013), entre outras, que irão disponibilizar uma infinidade de outras funções. Vale então a pena pesquisar bem antes de desenvolver a própria função, pois alguém já pode ter desenvolvido.

5.2 A forma geral de uma função

Como já mencionado, havendo uma situação em que é necessário reutilizar um código várias vezes, é conveniente que este código seja uma função. Em boa parte das situações, não haverá uma função pronta que atenda à necessidade específica, neste caso, será necessário desenvolvê-la. A linguagem Python permite definir funções e para isso, basta entender como elas funcionam e como aplicar estes conceitos ao problema computacional que pretende-se resolver. Veja a seguir a sintaxe para definir uma função.

```
1 #Sintaxe:
2 def nomeFuncao(parametro1, parametro2=padrao):
3     corpo da funcao
4     return valorRetornado
```

Na linha 2 da sintaxe, veja que, primeiro é necessário declarar a palavra reservada **def**, pois ela indica que será definida uma função neste trecho de código. Ainda na linha 2, veja que deve-se dar um nome à função, neste caso, substitua "nomeFuncao", por um nome que seja condizente com a ação da função, ou seja, um nome intuitivo. Para definir o nome da função, utilize as mesmas regras, já apresentadas na aula 1, ao definir o nome de variáveis.

Os parâmetros, ainda na linha 2, são o conjunto de valores que serão necessárias à função, para que a mesma, resolva o problema computacional em questão, ou seja, são os dados de entrada da função. Assim, é possível definir quantos forem necessários, os dados de entrada para que uma função resolva um problema. Por exemplo, imagine uma função para calcular a área de um retângulo, sabe-se que para fazer este cálculo é necessário a base e a altura do retângulo, assim, seria necessário definir estes dois parâmetros para esta função [1]. Seguindo o mesmo exemplo, como o objetivo é calcular

[1] Uma observação importante quanto aos parâmetros, é que, argumentos com valor padrão devem ser definidos após os argumentos sem valor padrão (BORGES, 2010, p. 52).

a área do retângulo, então é natural que o retorno dessa função seja o valor calculado, ou seja, essa será a saída.

Na linha 3, "corpo da funcao", deve ser escrito o código-fonte que se encarregará de resolver o problema para o qual a função está sendo criada. Naturalmente, o código-fonte não está limitado à 1 linha, podem ser utilizadas tantas quantas forem necessárias. O que define o início e término da função é a indentação, ou seja, segue o mesmo padrão adotado em outras cláusulas como o **if** e o **for**. Na linha 4, temos o "return valorRetornado", que será utilizado apenas quando faz parte do objetivo da função retornar um valor para o trecho de código que a invocou. A palavra **return** é fixa e a palavra "valorRetornado"deve ser substituída pelo valor que deseja-se retornar, que em geral, estará em uma variável. Veja a seguir um exercício para exemplificar.

5.2.1 Exercício de Exemplo

Faça um programa em Python que calcule a área de um retângulo, para isso o programa deve ler a altura e a base. O cálculo deve ser feito em uma função. Após calcular o programa deve imprimir o valor da área do retângulo.

```
1  #aqui comeca a definicao da funcao
2  def areaRetangulo(base, altura):
3      #calculo da funcao
4      area = base * altura
5      return area #retorno da funcao
6  #aqui a funcao terminou
7
8  #lendo os dados
9  vbase = float(raw_input("Informe a base do retangulo: \n"))
10 valtura = float(raw_input("Informe a altura do retangulo: \n"))
11
12 #invocando a funcao e retornado para a variavel
13 varea = areaRetangulo(vbase, valtura)
14
15 #imprimindo a variavel que recebeu o resultado
16 print "A area do retangulo e: ", varea
```

O programa apresentado é virtualmente separado em duas regiões, a primeira região inicia na linha 1 e vai até a linha 6, é onde se encontra a função de cálculo da área do retângulo, a segunda região inicia na linha 8 e vai até a linha 16, em que se encontra o programa principal, que é responsável pela execução da leitura dos valores da base, na variável *vbase*, e da altura, na variável *valtura*, e a invocação da função *areaRetangulo*, na linha 13, por fim, a impressão da área resultante do cálculo na linha 16. Veja que a função é invocada na linha 13, por meio da atribuição do retorno da função à variável *varea*, e na invocação da função são informados os dois argumentos necessários ao cálculo, por meio das variáveis *vbase* e *valtura*.

A figura 6 destaca o trecho do código-fonte da função, cada parte foi identificada para facilitar o entendimento. O nome da função é *areaRetangulo* e os argumentos são *base* e *altura*. Após o cálculo ser realizado na linha 4, o retorno do valor calculado é feito na linha 5. Veja também que na linha 4, são utilizadas as variáveis do argumento da função para o cálculo, todos os argumentos presentes na função podem ser utilizados como variáveis no corpo da função, e de fato, o objetivo dos argumentos é este, servir de entrada para os dados a serem processados.

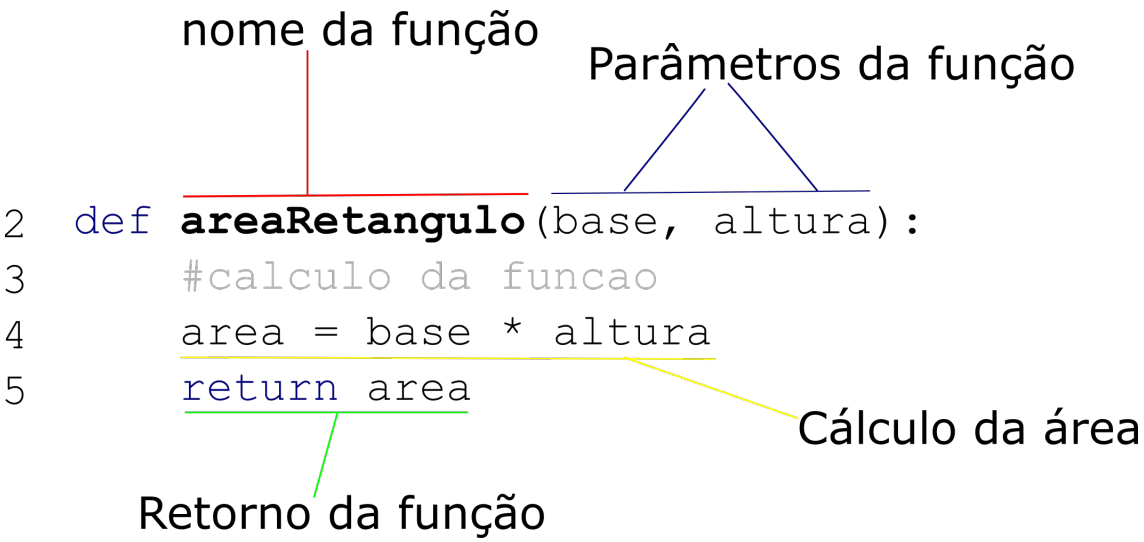

Figura 6 – Função de cálculo da área do retângulo

5.3 Definindo várias funções

Até o momento o nosso programa só tem uma função, mas assim como em outras linguagens, o Python possibilita ao programador definir quantas forem necessárias as funções em um programa, basta ter os devidos cuidados ao indentar o código, para que o interpretador seja capaz de delimitar corretamente aonde inicia e termina cada função, e o programa principal. Veja a seguir uma adaptação do mesmo exemplo, em que será adicionada a função para o cálculo da área do triângulo.

```
1  #aqui comeca a definicao da funcao do retangulo
2  def areaRetangulo(base, altura):
3      #calculo do retangulo
4      area = base * altura
5      return area #retorno da area
6  #aqui a funcao terminou a funcao do retangulo
7
8  #aqui comeca a definicao da funcao do triangulo
9  def areaTriangulo(base, altura):
10     #calculo do triangulo
11     area = (base * altura) / 2
12     return area #retorno da area
13 #aqui terminou a funcao do triangulo
14
15 #lendo os dados
16 vbase = float(raw_input("Informe a base do retangulo: \n"))
17 valtura = float(raw_input("Informe a altura do retangulo: \n"))
18
19 #invocando a funcao de calculo do retangulo
20 vareaRetangulo = areaRetangulo(vbase, valtura)
21 vareaTriangulo = areaTriangulo(vbase, valtura)
22
23 #imprimindo a a area do retangulo
24 print "A area do retangulo e: ", vareaRetangulo
25 print "A area do triangulo e: ", vareaTriangulo
```

Agora note que além função de cálculo da área do retângulo, entre as linha 9 e 12, foi adicionada também a função do cálculo da área do triângulo. Na linha 21, foi declarada uma variável para armazenar o resultado do cálculo da área do triângulo, e o nome da variável responsável por armazenar o resultado do cálculo da área do retângulo foi ajustada para manter o padrão na nomenclatura. Para ambos os cálculos são utilizados os mesmos valores, *base* e *altura*, assim, entre as linhas 15 e 17 não houve alterações. Na linha 21, foi adicionada a invocação para a função de cálculo da área do triângulo, note o nome da função para identificar as diferenças. Foi adicionada também a linha 25 para a impressão da área do triângulo.

Em relação à função de cálculo da área do triângulo, notou se há alguma semelhança com o cálculo da área do retângulo? Sim, há! Para calcular a área do triângulo, basta dividir o resultado do cálculo da área do retângulo por 2. Desta forma, podemos melhorar este código utilizando a função de cálculo da área do retângulo na função de cálculo da área do triângulo. Então podemos invocar uma função em outra função? Sim, veja a seguir os ajustes.

```
1  #aqui comeca a definicao da funcao do retangulo
2  def areaRetangulo(base, altura):
3      #calculo do retangulo
4      area = base * altura
5      return area #retorno da area
6  #aqui a funcao terminou a funcao do retangulo
7
8  #aqui comeca a definicao da funcao do triangulo
9  def areaTriangulo(base, altura):
10     #calculo do triangulo
11     area = areaRetangulo(base, altura) / 2
12     return area #retorno da area
13 #aqui terminou a funcao do triangulo
14
15 #lendo os dados
16 vbase = float(raw_input("Informe a base do retangulo: \n"))
17 valtura = float(raw_input("Informe a altura do retangulo: \n"))
18
19 #invocando a funcao de calculo do retangulo
20 vareaRetangulo = areaRetangulo(vbase, valtura)
21 vareaTriangulo = areaTriangulo(vbase, valtura)
22
23 #imprimindo a a area do retangulo
24 print "A area do retangulo e: ", vareaRetangulo
25 print "A area do triangulo e: ", vareaTriangulo
```

A única diferença para o código do exemplo anterior é a linha 11, note que a multiplicação da *base* por *altura* foi substituída pela invocação da função *areaRetangulo* e após esta função retornar o resultado da multiplicação, este será dividido por 2 para depois ser atribuído o resultado final à variável *area*.

5.4 Escopo das Variáveis

Entenda como sendo o escopo de uma variável, a área em que se poderá atribuir ou obter valores de uma determinada variável. Ou seja, a área em que a variável terá validade e significado. Basicamente, há dois tipos de escopo que uma variável pode abranger, local ou global. As variáveis globais podem ser acessadas de qualquer ponto

do código-fonte, ou seja, ela tem validade em qualquer função presente no código. Para que uma variável seja global, é necessário declarar a mesma fora das funções presentes no código. Ao alterar o valor de uma variável global, este valor será automaticamente válido em qualquer parte do código, desta forma, se duas ou três funções utilizam uma mesma variável global, então, ao obter o valor da variável, todas funções irão receber o mesmo valor (LUTZ; ASCHER, 2007).

As variáveis locais tem validade apenas na função em que foi declarada, ou seja, pode-se obter ou atribuir valores, apenas na função em que foram declaradas, assim, se o código-fonte possui, por exemplo, duas funções com variáveis locais com o mesmo nome, cada uma poderá assumir valores distintos em seu escopo (LUTZ; ASCHER, 2007). Até o momento, todos os códigos apresentados fizeram uso de variáveis locais. Veja a seguir um exemplo de uso de variáveis locais e globais.

```
1  #lendo os dados
2  vbase = float(raw_input("Informe a base do retangulo: \n"))
3  valtura = float(raw_input("Informe a altura do retangulo: \n"))
4
5  #aqui comeca a definicao da funcao do retangulo
6  def areaRetangulo():
7      #calculo do retangulo
8      area = vbase * valtura
9      return area #retorno da area
10 #aqui a funcao terminou a funcao do retangulo
11
12 #aqui comeca a definicao da funcao do triangulo
13 def areaTriangulo():
14     #calculo do triangulo
15     area = areaRetangulo() / 2
16     return area #retorno da area
17 #aqui terminou a funcao do triangulo
18
19 #invocando a funcao de calculo do retangulo
20 vareaRetangulo = areaRetangulo()
21 vareaTriangulo = areaTriangulo()
22
23 #imprimindo a a area do retangulo
24 print "A area do retangulo e: ", vareaRetangulo
25 print "A area do triangulo e: ", vareaTriangulo
```

No código apresentado como exemplo, foram realizados vários ajustes para mostrar o uso prático das variáveis globais e locais. As variáveis *vbase* e *valtura*, que antes haviam sido declaradas e lidas nas linhas 16 e 17, foram movidas para as linhas 2 e 3, para que as mesmas tenham escopo de global e sejam alimentadas antes da definição das funções, note que, ambas estão fora das funções, *areaRetangulo()* e *areaTriangulo()*, assim elas terão validade em todas essas funções. Note que nas linhas 6 e 13, o cabeçalho das funções *areaRetangulo()* e *areaTriangulo()* não têm mais os argumentos, pois como as variáveis *vbase* e *valtura* são globais, então os seus valores podem ser acessados em qualquer ponto, não sendo necessário a passagem do valor por referência como antes. Veja nas linhas 8, 24 e 25 como essas variáveis são acessadas, assim, basta utilizar elas normalmente sem se preocupar com o local. Agora observe que nas linhas 15, 20 e 21 as funções são invocadas sem a passagem dos parâmetros, pois como já foi dito, os valores das variáveis *vbase* e *valtura* são agora obtidos diretamente na função e não por referência como antes.

Por outro lado, algumas variáveis permaneceram como locais, como é o caso da variável *area* que é declarada nas duas funções, *areaRetangulo()* e *areaTriangulo()*, assim, o escopo dessas variáveis é local na função em que foram declaradas, e embora tenham o mesmo nome, ambas podem assumir valores diferentes, cada uma em seu escopo. É importante destacar que as variáveis locais de uma função não tem validade no escopo principal do programa, pois elas deixam de existir quando a função conclui a sua execução.

5.5 Retornando mais de um valor

É comum a necessidade de que a função retorne, ao programa que a invocou, mais de um valor. Geralmente as linguagens de programação permitem resolver essa necessidade com passagem por referência, ou objetos. No Python é possível resolver retornando um objeto com mais de um valor, como atributo, mas há uma forma mais elegante e simples de resolver. Veja a seguir um exemplo em formato de exercício.

5.5.1 Exercício de Exemplo

Faça um programa em Python que calcule os juros de um determinado saldo com base em uma taxa, ambos informados pelo usuário. O programa deve utilizar uma função para calcular os juros e retornar os juros e o saldo atual com base no saldo antigo acrescido dos juros. Ao final, a aplicação deve imprimir o total de juros e o novo saldo.

```
1  def calculaJuros(saldo, taxa):
2      juros = saldo * taxa / 100
3      saldo += juros
4      return juros, saldo
5
6  vSaldo = float(raw_input("Informe um saldo: "))
7  vTaxa = float(raw_input("Informe uma taxa: "))
8
9  vJuros, vSaldoNovo = calculaJuros(vSaldo, vTaxa)
10
11 print "Juros.....: ", vJuros
12 print "Saldo Novo: ", vSaldoNovo
```

Este exemplo traz uma situação que envolve o retorno de mais de um valor pela função. Note que entre as linhas 2 e 3 é que são realizados os cálculos de juros e novo saldo, na linha 4 o **return** foi configurado para retornar duas variáveis, *juros* e *saldo*, utilizando virgula (,) como separador. Se houvesse a necessidade de retornar 3, 4 ou mais valores, bastava seguir a mesma lógica, separando por virgula. Deve-se ter o cuidado ao invocar a função também, pois como ela está configurada para retornar mais de um valor, então, ao invocar, deve-se disponibilizar a mesma configuração de variáveis para que os valores sejam armazenados, como foi realizado na linha 9, veja que foram declaradas duas variáveis, *vJuros* e *vSaldoNovo* para receber os valores vindos da função.

5.6 Resumo da Aula

Na aula 5 foram apresentados conceitos que permitem definir nossas próprias funções. Foi destacado também a importância de antes de iniciar a implementação de uma nova função, fazer sempre uma pesquisa para identificar se o problema que se deseja resolver, já não tenha uma solução em uma função disponível.

Sobre a definição de funções, primeiro foi discutida a forma geral de uma função, aspectos como o retorno da função, o nome da função, a lista de parâmetros de uma função e o corpo da função, assim, para cada uma dessas características foram discutidas as regras gerais para a definição e o formato de uso. Foram apresentados alguns exemplos de uso de funções para esclarecer variados aspectos como, o uso da função e a declaração de variáveis locais e globais.

5.7 Exercícios da Aula

Os exercícios desta lista foram Adaptados de Lopes e Garcia (2002, p. 42-61; 400-438).

1. Faça um programa em Python que leia três números e, para cada um, imprimir o dobro. O cálculo deverá ser realizado por uma função e o resultado impresso ao final do programa.

2. Faça um programa que receba as notas de três provas e calcule a média. Para o cálculo, escreva uma função. O programa deve imprimir a média ao final.

3. Faça um programa em Python que leia o valor de um ângulo em graus e o converta, utilizando uma função, para radianos e ao final imprima o resultado. Veja a fórmula de cálculo a seguir.

$$rad = \frac{ang \times pi}{180} \tag{5.1}$$

   Em que:

   - rad = ângulo em radianos
   - ang = ângulo em graus
   - pi = número do pi

4. Faça um programa que calcule e imprima o fatorial de um número, usando uma função que receba um valor e retorne o fatorial desse valor.

5. Faça um programa que verifique se um número é primo por meio de um função. Ao final imprima o resultado.

6. Faça um programa que leia o saldo e o % de reajuste de uma aplicação financeira e imprimir o novo saldo após o reajuste. O cálculo deve ser feito por uma função.

7. Faça um programa que leia a base e a altura de um retângulo e imprima o perímetro, a área e a diagonal. Para fazer os cálculos, implemente três funções, cada uma deve realizar um cálculo especifico conforme solicitado. Utilize as fórmulas a seguir.

$$perimetro = 2 \times (base + altura) \tag{5.2}$$

$$area = base \times altura \tag{5.3}$$

$$diagonal = \sqrt{base^2 + altura^2} \tag{5.4}$$

8. Faça um programa que leia o raio de um circulo e imprima o perímetro e a área. Para fazer os cálculos, implemente duas funções, cada uma deve realizar um cálculo especifico conforme solicitado. Utilize as fórmulas a seguir.

$$perimetro = 2 \times pi \times raio \tag{5.5}$$

$$area = pi \times raio^2 \tag{5.6}$$

9. Faça um programa que leia o lado de um quadrado e imprima o perímetro, a área e a diagonal. Para fazer o cálculo, implemente três funções, cada uma deve realizar um cálculo especifico conforme solicitado. Utilize as fórmulas a seguir.

$$perimetro = 4 \times lado \tag{5.7}$$

$$area = lado^2 \tag{5.8}$$

$$diagonal = lado \times \sqrt{2} \tag{5.9}$$

10. Faça um programa que leia os lados *a*, *b* e *c* de um paralelepípedo e imprima a diagonal. Para fazer o cálculo, implemente uma função. Utilize a fórmula a seguir.

$$diagonal = \sqrt{a^2 + b^2 + c^2} \tag{5.10}$$

11. Faça um programa que leia a diagonal maior e a diagonal menor de um losango e imprima a área. Para fazer o cálculo, implemente uma função. Utilize a fórmula a seguir.

$$area = \frac{(diagonalMaior \times diagonalMenor)}{2} \tag{5.11}$$

12. Faça um programa que leia os catetos (dois catetos) de um triângulo retângulo e imprima a hipotenusa. Para fazer o cálculo, implemente uma função. Utilize a fórmula a seguir.

$$hipotenusa = \sqrt{cateto1^2 + cateto2^2} \tag{5.12}$$

13. Em épocas de pouco dinheiro, os comerciantes estão procurando aumentar suas vendas oferecendo desconto. Faça um programa que permita entrar com o valor de um produto e o percentual de desconto e imprimir o novo valor com base no percentual informado. Para fazer o cálculo, implemente uma função.

14. Faça um programa que verifique quantas vezes um número é divisível por outro. A função deve receber dois parâmetros, o dividendo e o divisor. Ao ler o divisor, é importante verificar se ele é menor que o dividendo. Ao final imprima o resultado.

15. Construa um programa em Python que leia um caractere (letra) e, por meio de uma função, retorne se este caractere é uma consoante ou uma vogal. Ao final imprima o resultado.

16. Construa um programa que leia um valor inteiro e retorne se a raiz desse número é exata ou não. Escreva uma função para fazer a validação. Ao final imprima o resultado.

17. Implemente um programa que leia uma mensagem e um caractere. Após a leitura, o programa deve, por meio de função, retirar todas as ocorrências do caractere informado na mensagem colocando * em seu lugar. A função deve também retornar o total de caracteres retirados. Ao final, o programa deve imprimir a frase ajustada e a quantidade de caracteres substituídos.

18. Faça um programa que leia um vetor com tamanho 10 de números inteiros. Após ler, uma função deve verificar se o vetor está ordenado, de forma crescente ou decrescente, ou se não está ordenado. Imprimir essa resposta no final do programa.

19. Faça um programa que leia um vetor com tamanho 10 de números inteiros. Após ler, uma função deve criar um novo vetor com base no primeiro, mas, o novo vetor deve ser ordenado de forma crescente. O programa deve imprimir este novo vetor após a ordenação.

20. Faça um programa que leia 20 de números inteiros e armazene em um vetor. Após essa leitura, o programa deve ler um novo número inteiro para ser buscado no vetor. Uma função deve verificar se o número lido por último está no vetor e retornar a posição do número no vetor, caso esteja, ou -1, caso não esteja.

AULA 6

Arquivos

Metas da Aula

1. Aprender a lidar com arquivos na linguagem Python.
2. Aplicar variadas situações relacionadas ao uso de arquivos em programação.
3. Aprender a escrever programas em linguagem Python que façam uso de arquivos.

Ao término desta aula, você será capaz de:

1. Escrever programas que manipulam arquivos.
2. Escrever programas que usam informações provenientes de arquivos.

6.1 Porque manipular arquivos?

Ao lidar com aplicações científicas, como é o caso dos programas que utilizam heurísticas e meta-heurísticas para resolver problemas de otimização, é comum precisarmos lidar com arquivos, que em geral, estão em formato texto plano ou bruto. Mas, o que seria essa texto plano ou bruto? Refere-se aos arquivos que armazenam apenas o texto sem nenhum elemento adicional, como formatadores, imagens, tabelas, etc. Em geral, estes arquivos tem o formato "txt"ou "csv".

Mas, porque precisamos desses arquivos? Porque geralmente precisamos manipular uma quantidade grande de dados, então é viável incluir esses dados em um determinado arquivo, pois assim, qualquer programa poderá ser preparado para ler as informações a partir do arquivo, isso garantirá uma generalização das informações, caso seja, por exemplo necessário alterar os dados, ou mesmo alterar a linguagem de programação.

Para exemplificar, imagine a seguinte situação: Você fez um programa de computador para resolver um problema de otimização com heurísticas, e você preparou este programa para ler os dados a partir de um arquivo que contém informações de um determinado problema para um determinado período de tempo, exemplo: um ano, um semestre ou um mês. Agora você deseja analisar qual seria o resultado se aplicar o mesmo programa em dados da mesma origem, contudo, de um período diferente. Será muito fácil não é mesmo? Claro, pois como você já preparou o programa para ler os dados a partir de um arquivo, então basta obter um novo arquivo com o mesmo formato e os dados que deseja para o novo período.

Por outro lado, se você 'embutir' todas as informações necessárias ao problema diretamente no código-fonte do programa, o esforço de alterar a fonte dos dados será imenso. Da mesma forma, se você desejar alterar a linguagem de programação, também terá muito trabalho. Então, podemos concluir que, de fato, o uso de arquivos é muito importante para facilitarmos o reuso dos recursos, seja o programa ou os dados.

6.2 Lendo arquivos

Como já mencionado antes, a linguagem Python nos disponibiliza várias funções que permitem executar tarefas comuns a muitos problemas, isso não é diferente com a manipulação de arquivos (CRUZ, 2017, p. 165), assim, podemos utilizar as próprias funções do Python que permitem manipular o arquivo, que neste caso são: **open()**, **readline()** e **close()**. O uso destas funções é bem simples. Para testar, vamos criar um arquivo de exemplo, siga os passos:

Passos

1. Crie um diretório em seu computador conforme: **C:\AulaPython**;
2. Copie os dados disponibilizados a seguir;
3. Cole em um editor de textos plano, como o **notepad**, **notepad++**, ou outro de sua preferência;
4. Salve o arquivo com o nome: **dados1Aula.txt** na pasta criada, pois vamos referenciar ela em nossos testes.

Dados para serem salvos no arquivo

```
4 2
10 7 6 9
5 3 4 8
10 5
```

Fonte: Dados fictícios

Suponhamos que os dados disponibilizados são relativos ao problema clássico da mochila, assim, a primeira linha refere-se ao número de objetos, o valor 4 e ao número de mochilas, o valor 2. A linha 2 refere-se aos valores dos objetos, como são 4 objetos, então temos 4 valores. A linha 3 refere-se ao peso dos objetos, por fim, a linha 4 refere-se à capacidade das mochilas, como são duas mochilas, então são 2 valores para a capacidade das mochilas. Assim, temos um exemplo em que os dados estão organizados, mas não tem uma estrutura homogênea, pois irá variar de acordo com o número de objetos e o número de mochilas. Veja a seguir o exemplo lendo a primeira linha:

```
1  >>> f = open(r"C:\AulaPython\dados1Aula.txt", "r")
2  >>> linha = f.readline()
3  >>> valores = linha.split()
4  >>> valores
5  ['4', '2']
6  >>> numObjetos = int(valores[0])
7  >>> numMochilas = int(valores[1])
8  >>> numObjetos
9  4
10 >>> numMochilas
11 2
```

Na primeira linha foi utilizado o comando **open()** para abrir o arquivo, note que o comando recebeu dois argumentos, o primeiro é o endereço e o nome do arquivo que deve ser lido, o segundo é o modo de abertura que pode variar conforme a tabela 8, no caso, como a intenção é abrir o arquivo em modo de leitura, foi utilizado a letra **"r"**. Observe ainda que o resultado da abertura foi atribuído ao objeto **"f"**, esse passa a ser o objeto de acesso ao arquivo, conforme pode ser visto nas linhas seguintes.

Tabela 8 – Modos de abertura de arquivos em Python

Modo de Abertura	**Descrição**
'r'	Abre em modo de leitura (padrão)
'w'	Abre em modo de escrita, truncando o arquivo primeiro
'x'	Abre exclusivamente para criação, falhando se já existir
'a'	Abre para escrita, adicionando a partir do final do arquivo se já existir
'b'	Abre em modo binário
't'	Abre em modo texto (padrão)
'+'	Abre o arquivo do disco para atualização (lendo e escrevendo)

Fonte: Adaptado de (CRUZ, 2017, p. 167)

Na linha 2 do código foi utilizada a função **readline()** para obter a primeira linha, essa função obtêm sempre a próxima linha a ser lida a cada vez que ela é acionada, como nenhuma linha foi lida ainda, então ela irá retornar a primeira. O resultado é salvo nessa variável denominada como *linha*. Como sabemos os valores presentes em

cada linha são separados por espaço em branco, assim, convém utilizar o comando **split()**, para obter apenas os valores presentes na linha e separados em um vetor. Por isso, foi utilizado esse comando na linha 3 e os valores obtidos foram atribuídos ao vetor *valores*. Conforme pode se ver nas linhas 4 e 5, o vetor *valores* foi preenchido conforme esperado.

Como já temos os valores separados da primeira linha, agora é só atribuir aos objetos devidamente, como o primeiro valor refere-se ao número de objetos para serem atribuídos às mochilas, então na linha 6 foi definida a variável *numObjetos* para receber este valor. Da mesma forma, na linha 7 foi definida a variável *numMochilas* para receber o valor referente ao número de mochilas. Para validar se deu certo, a linha 8 exibe o valor de *numObjetos* e a linha 10 exibe o valor de *numMochilas*. Na sequência vamos ler as próximas linhas, o código de exemplo é uma continuação do anterior.

```
1  >>> linha = f.readline()
2  >>> valores = linha.split()
3  >>> valores
4  ['10', '7', '6', '9']
5  >>> vetValoresObjetos = []
6  >>> for val in valores:
7      vetValoresObjetos.append(int(val))
8
9  >>> vetValoresObjetos
10 [10, 7, 6, 9]
```

Para ler a linha 2 foi utilizado o mesmo procedimento, na primeira linha o comando **readline()** para atribuir a segunda linha à variável *linha*. Na linha 2 foi utilizado o mesmo comando **split()** para separar os valores. Na linha 5 foi declarado o vetor *vetValoresObjetos* para armazenar os valores dos objetos e nas linhas 6 e 7 foi executado um laço **for** pelos valores da linha e, a cada valor lido, o mesmo é convertido para inteiro e adicionado ao vetor *vetValoresObjetos*. O próximo código mostra a leitura das linhas seguintes.

```
1  >>> linha = f.readline()
2  >>> valores = linha.split()
3  >>> valores
4  ['5', '3', '4', '8']
5  >>> vetPesosObjetos = []
6  >>> for val in valores:
7      vetPesosObjetos.append(int(val))
8
9  >>> vetPesosObjetos
10 [5, 3, 4, 8]
11 >>> linha = f.readline()
12 >>> valores = linha.split()
13 >>> vetCapacidadeMochilas = []
14 >>> for val in valores:
15     vetCapacidadeMochilas.append(int(val))
16
17 >>> vetCapacidadeMochilas
18 [10, 5]
19 >>> f.close()
```

Entre as linhas 1 e 7 foi lida a próxima linha que se refere aos pesos dos objetos e entre as linhas 11 e 15 foi lida a linha referente à capacidade das mochilas, observe que

foi utilizada para ambas a mesma estratégia de leitura anterior. Por fim, na linha 19 o arquivo é fechado, é sempre importante executar essa ação ao final de qualquer leitura de arquivo.

Esse método de realizar a leitura de arquivos é, de certa forma, trabalhoso, mas necessário, pois como mencionado o arquivo do exemplo não possui uma estrutura homogênea, sendo assim necessária a leitura de cada linha atribuindo a variáveis distintas conforme o valor em questão. Bem, contudo, há situações em que o arquivo a ser lido terá uma estrutura homogênea, geralmente em formato de tabela, neste caso há bibliotecas extras, como o **pandas** e **numPy** que disponibilizam os mesmos recursos, mas, sendo mais eficientes em muitos aspectos (MCKINNEY, 2013, p. 155).

Geralmente, arquivos com estrutura homogênea são do tipo "csv", um tipo clássico de arquivos de texto que funcionam com delimitadores, para determinar a separação das colunas dos dados presentes no arquivo. Para manipular arquivos "csv", vamos utilizar a biblioteca **NumPy**, que disponibiliza a função **genfromtxt()** que é especifica para manipular arquivos do tipo texto (MCKINNEY, 2013, p. 104). Veja a seguir a sintaxe:

```
1 #sintaxe:
2 import numpy as np
3 nomeDados = np.genfromtxt(r"diretorioEnomeDoArquivo.extensao", delimiter=",",
    dtype=None, skip_header=0, names=('coluna1', 'coluna2', ..., 'colunaN'))
```

Conforme a sintaxe, primeiro é necessário importar a biblioteca **NumPy**, na linha 2, veja que foi dado um apelido à biblioteca ao importá-la, no caso "np", após isso, qualquer função da biblioteca deve ser precedida pelo apelido. Na linha 3, definiu-se o nome para instanciar o arquivo, este nome será a referência para o programador fazer acessos ao arquivo posteriormente. Na sequência, por meio de atribuição, deve-se invocar a função **genfromtxt()**, que embora possa receber vários argumentos, no geral, poderá ser configurada satisfatoriamente com pelo menos quatro dos argumentos apresentados na sintaxe, note que a invocação da função é precedida pelo apelido. O primeiro argumento é o diretório em que se encontra o arquivo que deve ser lido e o nome do arquivo.

A partir do segundo argumento todos são opcionais, ou seja, não há a necessidade de informar, contudo serão assumidos os valores padrões. O segundo argumento é o delimitador, que permite especificar qual é o separador entre as colunas presentes nos arquivos. É natural que arquivos de dados tenham várias colunas e que haja um delimitador para separá-las, por exemplo, tabulação, virgula ou ponto e virgula. Caso você não informe este parâmetro, o separador padrão é uma sequência de espaços em branco.

O terceiro argumento permite definir o tipo ao importar as colunas, podemos sempre utilizar "None", pois assim, o **genfromtxt()** se encarregará de definir o tipo de cada coluna conforme os dados encontrados. O quarto parâmetro, **skip_header**, permite indicar quantas linhas você deseja que o **genfromtxt()** ignore ao fazer a importação. É importante destacar que ele ignora apenas linhas do início. Caso não seja informado, o valor padrão é zero. O quinto parâmetro permite definir os nomes das colunas que estão sendo importadas. Para exemplificar, vamos criar um pequeno arquivo "csv"com algumas informações para testarmos a manipulação de arquivos. Para isso, execute as seguintes etapas:

Passos

1. Copie os dados disponibilizados a seguir;
2. Cole em um editor de textos plano, como o **notepad**, **notepad++**, ou outro de sua preferência;
3. Salve o arquivo com o nome: **dados2Aula.csv** na pasta criada, **C:\AulaPython**, que já foi criada anteriormente.

Dados para serem salvos no arquivo

Matricula;Nome;Idade;Nota1;Nota2;Nota3
1;Paulo Souza;16;6.6;7.3;8.0
2;Marcos Reis;14;7.8;8.2;8.1
3;Sandra Oliveira;15;1.5;6.9;8.5
4;Andre Sobreiro;16;5.5;9.1;7.5
5;Paula Mendes;14;8.8;4.6;9.0
7;Silvia Mosa;15;7.2;4.3;7.7
8;Carlos Andrade;14;8.1;8.8;7.0
9;Darlene Costa;16;2.1;5.5;4.2
10;Mateus Moraes;16;9.0;9.5;9.2

Fonte: Dados fictícios

Como pode ver os dados são fictícios de 10 alunos, com matrícula, nome, idade e 3 notas. Dez alunos é uma pequena quantidade de dados, mas é mais que suficiente para aprendermos a manipular os arquivos. Observe que foi utilizado um separador em cada informação, no caso, ponto e vírgula (**;**). Embora olhando assim rapidamente o arquivo pareça ser confuso e bagunçado, a função **genfromtxt()** fará a leitura sem dificuldades. Veja a seguir o exemplo:

```
1  >>> import numpy as np
2  >>> dadosAlunos = np.genfromtxt(r"C:\AulaPython\dados2Aula.csv", delimiter=";",
       dtype=None, skip_header=1, names=('Matricula', 'Nome', 'Idade', 'Nota1', '
       Nota2', 'Nota3'))
3  >>> dadosAlunos
4  array([( 1, 'Paulo Souza', 16,  6.6,  7.3,  8. ),
5         ( 2, 'Marcos Reis', 14,  7.8,  8.2,  8.1),
6         ( 3, 'Sandra Oliveira', 15,  1.5,  6.9,  8.5),
7         ( 4, 'Andre Sobreiro', 16,  5.5,  9.1,  7.5),
8         ( 5, 'Paula Mendes', 14,  8.8,  4.6,  9. ),
9         ( 7, 'Silvia Mosa', 15,  7.2,  4.3,  7.7),
10        ( 8, 'Carlos Andrade', 14,  8.1,  8.8,  7. ),
11        ( 9, 'Darlene Costa', 16,  2.1,  5.5,  4.2),
12        (10, 'Mateus Moraes', 16,  9. ,  9.5,  9.2)],
13       dtype=[('Matricula', '<i4'), ('Nome', 'S15'), ('Idade', '<i4'), ('Nota1', '
           <f8'), ('Nota2', '<f8'), ('Nota3', '<f8')])
```

Na linha 1 ocorre a importação do **NumPy** conforme já mencionado. Na linha 2 a leitura dos dados com a função **genfromtxt()**, observe que os dados serão importados para o objeto de nome *dadosAlunos*, que será uma lista com várias colunas de informações heterogêneas. No primeiro argumento foi informado o diretório e o nome do arquivo conforme definimos para o exemplo. No segundo argumento foi informado o

separador **";"**, pois é utilizado no arquivo "csv", no terceiro argumento foi informado o tipo "None"e conforme pode ser visto na linha 13, cada coluna teve o tipo determinado conforme os dados encontrados, exemplo: a coluna 'Matricula' foi definida com **i4**, ou seja, **i** de inteiro e **4** de 4 bytes.

No quarto argumento foi informado 1, pois como o nosso arquivo tem um cabeçalho com o nome das informações nas colunas, é importante 'pular' essa linha, pois se ela não for ignorada, todas as colunas terão o tipo definido como **string**, pois a primeira informação encontrada será um texto. Por fim, no quinto argumento foram informados os nomes de cada coluna, essa informação poderia ser ignorada, a única diferença é que a própria função iria criar os nomes e estes não seriam intuitivos.

Observe agora a linha 3, ao digitar *dadosAlunos* o interpretador mostrou os dados presentes nesta lista. Veja que todos os dados foram corretamente importados, separados por coluna e convertidos conforme seu tipo. De posse dos dados, basta fazer uso desta lista conforme os conceitos aprendidos na aula 4.

6.3 Salvando arquivos

Quando estamos implementando aplicações para resolver problemas de otimização é comum precisarmos salvar informações em arquivo relacionadas aos resultados encontrados, como: a melhor solução encontrada, o valor da função objetivo, o tempo para encontrar a melhor solução, entre outras informações. Novamente a biblioteca **NumPy** possui uma função que facilita o trabalho de salvar arquivos, é a **savetxt()** que permite salvar uma lista no arquivo texto, assim, basta ao programador montar uma lista com as informações que deseja salvar e utilizar a função **savetxt()** para salvar em disco (MCKINNEY, 2013, p. 104). Veja a sintaxe a seguir:

```
1 #sintaxe:
2 import numpy as np
3 np.savetxt(r"diretorioEnomeDoArquivo.extensao", lista,  fmt='%s', delimiter=",",
    newline='\n', header='', footer=''))
```

Similar ao **genfromtxt()** o **savetxt()** também possui vários argumentos que são opcionais. o primeiro argumento é obrigatório, pois é a especificação do nome do arquivo e local onde o mesmo deverá ser salvo. O segundo argumento, também obrigatório, é a lista que pretende-se salvar, o terceiro argumento é importante na maioria dos casos, pois a menos que todos os dados presentes na lista sejam do mesmo tipo, então será necessário especificar o formato. O quarto argumento tem a mesma função de separador da função **genfromtxt()**, o quinto elemento é útil quando a lista possui, no elemento, um delimitador de quebra de linha, como o '**\n**', o sexto permite especificar o se que deseja incluir no cabeçalho do arquivo e o sétimo permite especificar o que se deseja incluir no final do arquivo, ambos também são opcionais. Além destes, há outros argumentos que podem ser analisados no manual do **NumPy**. Veja a seguir um exemplo:

```
1 >>> dadosAlunos
2 array([( 1, 'Paulo Souza', 16,  6.6,  7.3,  8. ),
3        ( 2, 'Marcos Reis', 14,  7.8,  8.2,  8.1),
4        ( 3, 'Sandra Oliveira', 15,  1.5,  6.9,  8.5),
5        ( 4, 'Andre Sobreiro', 16,  5.5,  9.1,  7.5),
6        ( 5, 'Paula Mendes', 14,  8.8,  4.6,  9. ),
7        ( 7, 'Silvia Mosa', 15,  7.2,  4.3,  7.7),
8        ( 8, 'Carlos Andrade', 14,  8.1,  8.8,  7. ),
9        ( 9, 'Darlene Costa', 16,  2.1,  5.5,  4.2),
```

```
10        (10, 'Mateus Moraes', 16,  9. ,  9.5,  9.2)],
11       dtype=[('Matricula', '<i4'), ('Nome', 'S15'), ('Idade', '<i4'), ('Nota1', '
             <f8'), ('Nota2', '<f8'), ('Nota3', '<f8')])
12 >>> np.savetxt(r"C:\AulaPython\teste.txt", dadosAlunos, fmt='%s')
```

No exemplo, foi utilizado a lista importada do arquivo "csv"para salvar em formato texto. Observe a linha 12 que a função **savetxt()** não retorna valor, pois ela simplesmente salva o arquivo. Observe também que o primeiro parâmetro informa o mesmo diretório que já foi definido e informa o nome do arquivo em formato "txt", o segundo parâmetro informa a lista *dadosAlunos*, o terceiro parâmetro indica que os dados devem ser formatados como string. O *"%s"* foi definido com base na tabela 5 vista na aula 1. Após executar o comando o arquivo foi salvo conforme a figura 7. Observe na figura que, como não foi especificado nenhum delimitador, então as colunas foram separadas apenas por espaço em branco.

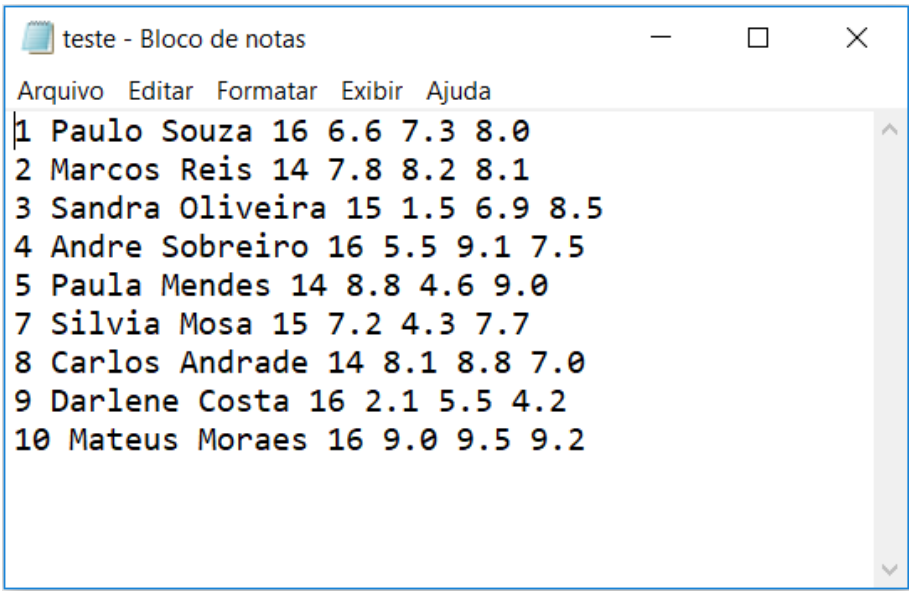

Figura 7 – Arquivo salvo com **salvetxt()**

Como a função **savetxt()** requer uma lista para o salvamento, talvez o seu uso não seja muito viável quando o que desejamos escrever no arquivo é totalmente desestruturado. Neste caso, uma alternativa simples para escrever em arquivos seria utilizar as próprias funções do Python que permitem manipular o arquivo, que neste caso são: **open()**, **write()** e **close()**. O uso destas funções é bem simples, veja a seguir um exemplo:

```
1 >>> f = open(r"C:\AulaPython\testenovo.txt", "w")
2 >>> f.write("Escrevendo a primeira linha em nosso arquivo\n")
3 >>> f.write("Escrevendo a segunda linha em nosso arquivo\n")
4 >>> f.close()
```

Observe que, com apenas 4 comandos foi possível salvar um arquivo no disco com duas linhas escritas. A primeira linha é responsável por abrir o arquivo com o comando **open()**, que por sua vez, recebeu dois parâmetros, o primeiro é o nome do arquivo e o diretório em que deve ser salvo, o segundo é o modo de abertura do arquivo, como a nossa intenção é escrever informações no arquivo então temos que abri-lo em modo de escrita, por isso foi informada a letra "w", conforme a tabela 8 já apresentada.

As linhas 2 e 3 do código-fonte de exemplo foram responsáveis por escrever o texto no arquivo, observe que foi incluso o **\n** ao final de cada linha para que ocorra a quebra de linha no arquivo de texto. Por fim, a linha 4 é responsável por fechar o arquivo. Este passo é muito importante, pois se o programador não fechar o arquivo no momento correto, o mesmo provavelmente será corrompido, a figura 8 representa o arquivo após o salvamento.

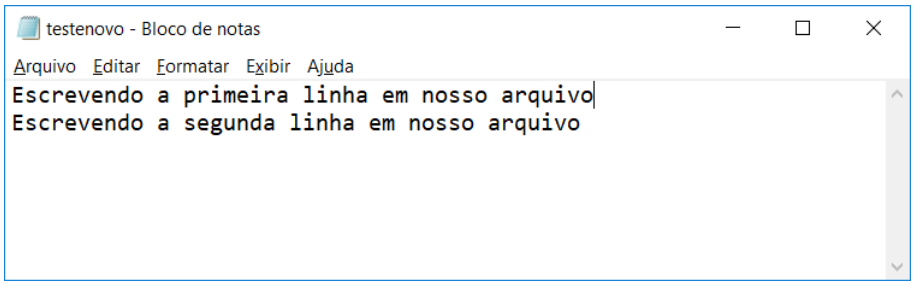

Figura 8 – Arquivo salvo com os comandos do Python

Observe na tabela 8 que uma alternativa é utilizar a letra "a"para modo de abertura, assim, fica fácil, por exemplo, abrir, escrever, fechar e depois abrir de novo para escrever novamente no arquivo, pois abrindo com este modo a escrita será sempre no final do arquivo e se o arquivo já existir, a função **open()** vai abri-lo, caso contrário, vai cria-lo, de forma que o programador não precisará se preocupar em fazer este controle. Esse recurso nos permite então escrever facilmente em arquivos de forma não estruturada e com poucas linhas de código.

6.4 Resumo da Aula

Nesta aula foram apresentados os principais conceitos sobre como manipular arquivos em Python, conceitos estes que serão necessários para dar continuidade no estudo, pois no geral, quando trabalhamos com heurísticas para resolver problemas de otimização, é comum precisarmos manipular arquivos, como fazer a leitura de um arquivo, ou escrever os resultados em um arquivo.

Assim, para ler um arquivo, aprendemos a utilizar as funções nativas do Python, como: **open()**, **readline()** e **close()**. Essas funções tem uma manipulação um pouco mais trabalhosa, contudo, permitem a leitura de arquivos desestruturados. Para a leitura de arquivos estruturados, foi apresentada a função **genfromtxt()** da biblioteca **NumPy**, pois como mencionando, poderíamos utilizar as funções nativas do Python para fazer essa operação, contudo, essa função traz um benefício muito grande no formato em que ela entrega os dados para o programador, pois elimina o trabalho que o programador teria de tratar a informação se o mesmo fosse utilizar as funções nativas do Python.

Para salvar o arquivo, aprendemos a função **salvetxt()**, também da biblioteca **NumPy** e as funções nativas do Python também foram apresentadas, pois neste caso, vimos que a função **salvetxt()** é muito útil quando os dados estão estruturadas em uma lista, mas nem sempre temos informações de resultados estruturada, neste caso, pode ser melhor utilizar as funções nativas.

6.5 Exercícios da Aula

1. Faça um programa em Python que leia um arquivo de sua escolha mas que seja do tipo "csv". Após obter os dados do arquivo, obtenha a quantidade de registros do arquivo com os conceitos aprendidos na aula 4.

2. Faça um programa que leia um arquivo texto qualquer e após isso, adicione duas novas linhas de texto no arquivo e salve no disco. O arquivo que você escolher deve ser do tipo "txt".

3. Faça um programa em Python que leia os dados de produto: código, nome, quantidade e preço. O programa deve ser capaz de ler os dados enquanto o usuário desejar, para isso utilize os conceitos aprendidos na aula 3. Após cada produto ser cadastrado pelo usuário, adicione os dados dele em um arquivo que deve ser salvo no disco. Ao encerrar o laço de cadastros, o arquivo deve ser fechado. Neste exercício você deverá utilizar as funções nativas do Python.

4. Refaça o exercício anterior, mas desta vez salvando os dados em um lista, depois utilize a função **salvetxt()** para salvar a lista no disco. Ao configurar a função **salvetxt()** defina o arquivo como "csv"e defina um delimitador=**";"**.

5. Altere o programa construído no exercício anterior para que leia o arquivo produzido no exercício anterior, e após o usuário cadastrar novos produtos, que serão adicionados na lista já lida, salve novamente o arquivo no disco.

AULA 7

Números Aleatórios

Metas da Aula

1. Compreender a necessidade dos números aleatórios em Métodos heurísticos para Otimização.
2. Aprender a gerar números pseudo-aleatórios em Python.

Ao término desta aula, você será capaz de:

1. Determinar quando será necessário produzir os números aleatórios.
2. Gerar números pseudo-aleatórios em Python.

7.1 Porque precisamos de números aleatórios ao utilizar métodos heurísticos para resolver problemas de otimização?

Esse entendimento é algo que talvez não fique muito claro neste momento, contudo é importante fazer um esclarecimento, mesmo que superficial, para abordar esta técnica na parte I do livro. Ao longo da parte II, este conceito será melhor esclarecido, preenchendo assim, as lacunas que restarem. Para facilitar o entendimento, suponha que existam dois tipos de métodos para resolver problemas de otimização, os **métodos exatos** e os **métodos aproximativos**. Os métodos exatos surgiram primeiro e basicamente o seu objetivo é: avaliar todas as possibilidades de combinação dos fatores que se pretende otimizar para obter a solução ótima.

Peguemos o exemplo do problema do caixeiro viajante. Neste problema o caixeiro viajante deve viajar, passando por várias cidades, para visitar os seus clientes, algumas cidades possuem mais de um possível caminho ou rota, assim o objetivo deste problema é otimizar a rota de tal forma que o caixeiro viajante percorra a menor quantidade de quilómetros, obtendo assim o menor custo. Este problema tem alta complexidade em função do número de combinações possíveis que devem ser avaliadas pelo método exato.

O número de combinações possíveis é dado por *(n-1)!*, assim, se o caixeiro viajante tiver que visitar 4 cidades, o número de possíveis rotas é *3!*, ou seja, 6 possíveis rotas. Mas, se aumentarmos para 5 cidades, então seriam 24 possíveis rotas, e se aumentarmos para 6 cidades, neste caso, o caixeiro terá que calcular as 120 possíveis rotas, e se elevarmos este número para 7 cidades, o número de rotas aumenta para 720. Percebeu? Apenas 7 cidades e o cálculo já se torna difícil de avaliar todas as possibilidades, aumentando para 50 cidades teremos o seguinte número de possibilidades: $(6.08281864 \times 10)^{62}$, o que provavelmente levará um computador atual a gastar anos para examinar todas as possibilidades. Este é um dos muitos problemas que são do tipo NP-difícil, que na prática quer dizer que é um problema de difícil solução por métodos exatos.

Considerando este problema na utilização dos métodos exatos é que surgiram os métodos aproximativos, que na prática não conseguem obter a solução ótima, como é o caso dos métodos exatos, contudo, os métodos aproximativos, de uma forma geral, conseguem obter soluções muito próximas da solução ótima com a vantagem de fazerem isso em um tempo muito menor (GOLDBARG; GOLDBARG; LUNA, 2016, p. 46). Mas, como as heurísticas, enquanto soluções aproximativas conseguem obter soluções de boa qualidade com muito menos esforço que os métodos exatos? É ai que entram os números aleatórios, pois é comum que um dos elementos presentes nas heurísticas seja o sorteio aleatório de uma determinada solução.

O sorteio de uma solução é necessário justamente porque os métodos heurísticos não irão examinar todas as combinações, tampouco irão examinar as mesmas combinações em todas as execuções, pois neste caso, não haveria eficiência na busca da solução de boa qualidade. Desta forma, esta aula tem o objetivo de abordar os conceitos necessários para produzir os números aleatórios em Python, pois este conhecimento será necessário nas próximas aulas.

7.2 Biblioteca **random** do Python

A biblioteca **random()** disponível no Python permite gerar números pseudo-aleatórios em vários formatos. Mas, porque pseudo-aleatórios? Porque um computador não é capaz de realmente gerar números aleatórios, por isso, dizemos pseudo-aleatórios

(DOWNEY; ELKNER; MEYERS, 2010, p. 87). Assim, geralmente as linguagens utilizam uma estratégia de gerar o número aleatório com base em uma semente, para garantir que serão sempre sequências de números sorteados diferentes à cada execução do programa. O Python também utiliza esta estratégia, para isso, disponibiliza a função **seed()** pertencente à **random** (MCKINNEY, 2013, p. 107). A função **seed()** deve ser acionada uma vez a cada execução do programa. Vejamos a seguir um primeiro exemplo:

```
1 >>> from random import *
2 >>> seed()
3 >>> print random()
4 0.549489889896
5 >>> print random()
6 0.298710249508
7 >>> numero = random()
8 >>> print numero
9 0.402121471943
```

Na primeira linha, veja que foi necessário importar a biblioteca **random**, note que o * ao fim da instrução indica que desejamos importar todas as funções da biblioteca. Esta importação só precisa ser realizada uma vez no ciclo de vida da aplicação. Na linha 2, após importar **random** foi acionado **seed()** para gerar a semente que vai garantir os números aleatórios. Da mesma forma, **seed()** só precisa ser acionado uma vez durante a execução do programa.

Na linha 3 a função **random()**, responsável por gerar os números aleatórios, foi invocada pela função **print** e o resultado é apresentado na linha 4, veja que é um número real, sim, por padrão a função **random()** produz números reais entre **0 e 1**. Observe agora na linha 5, foi acionado novamente o comando e na linha 6 foi produzido um novo número aleatório. É importante destacar que podemos também atribuir os valores gerados por **random()** à uma variável, conforme as linhas 7 à 9. Uma observação importante! Se você executar os mesmos comandos em seu computador, é muito pouco provável que obtenha os mesmos valores que obtive no exemplo, pois como se tratam de números aleatórios a chance deles serem iguais é muito pequena.

7.3 Gerando números aleatórios entre uma faixa de valores

Como vimos, por padrão a função **random()** gera números aleatórios reais entre 0 e 1, mas e se precisarmos gerar números entre uma determinada faixa de valores, por exemplo, entre 1 e 10, como proceder? Neste caso, iremos utilizar a função **uniform()**, também pertencente à biblioteca **random**, veja a seguir:

```
1 >>> print uniform(1, 10)
2 2.03043837483
3 >>> print uniform(1, 10)
4 6.97068054804
5 >>> numero = uniform(1, 10)
6 >>> print numero
7 5.03735762412
```

No exemplo apresentado, não foi realizada a importação de **ramdom** e não foi definida a semente, pois já foi realizado antes. Na linha 1 foi invocada a função **uniform()** com os argumentos 1 e 10, para que sejam gerados números aleatórios reais entre 1 e

10, conforme pode ser visto na linha 2. Veja que ao acionar novamente foi gerado um novo número aleatório conforme a linha 4, e da mesma forma que **random()**, também podemos atribuir o valor gerado por **uniform()** à uma variável, conforme a linha 5.

7.4 Gerando números aleatórios inteiros

Até o momento nós geramos apenas números reais, mas é muito provável que vamos precisar gerar números aleatórios pertencentes ao conjunto dos inteiros. Neste caso, temos a função **randint()**, que também pertence à biblioteca **random**. Veja a seguir o exemplo de como usar:

```
1 >>> print randint(1, 100)
2 88
3 >>> print randint(1, 100)
4 75
5 >>> numero = randint(1, 100)
6 >>> print numero
7 90
```

Conforme o exemplo, a função **randint()** segue os mesmos padrões das funções já vistas, pois recebe argumentos e podemos atribuir o número gerado à uma variável, conforme a linha 5. Para utilizar a função **randint()**, informe os dois argumentos que irão estabelecer a faixa de valores que deseja para os números aleatórios. No exemplo, a faixa estabelecida foi 1 a 100 e conforme podemos ver nas linhas 2, 4 e 7, os números gerados ficaram todos entre esta faixa.

7.5 Números aleatórios com a função **randrange()**

Imagine que, por algum motivo, você precise que o Python gere números aleatórios inteiros, mas que sejam apenas números ímpares, ou apenas números pares, ou ainda que sejam múltiplos de 10. Isso é possível com a função **randrange()**. Veja alguns exemplos de uso a seguir:

```
1  >>> print randrange(1, 1000, 2)
2  231
3  >>> print randrange(1, 1000, 2)
4  455
5  >>> print randrange(1, 1000, 2)
6  1
7  >>> print randrange(2, 1000, 2)
8  530
9  >>> print randrange(0, 1000, 10)
10 660
```

Nas linhas 1, 3 e 5 do exemplo foi invocada a função **randrange()** com os argumentos **1** como valor inicial da faixa, **1000** como valor final da faixa e **2** como salto, ou seja, os números serão gerados saltando de 2 em 2 e começando de 1, assim, pode-se observar nas linhas 2, 4 e 6 que foram gerados números inteiros ímpares. Na linha 7, foi modificado apenas o início para **2**, e assim, foram gerados números pares. Na linha 9 a função teve modificado os parâmetros: início para **0** e o passo para 10, gerando assim números múltiplos de 10.

7.6 Resumo da Aula

Nesta aula foram apresentados os principais conceitos sobre como gerar números aleatórios em Python, conceitos estes que serão necessários para dar continuidade no estudo. Pois, boa parte dos métodos aproximativos, utilizam a estratégia de produzir números aleatórios para examinar muitas combinações, sem ter que avaliar todas as possibilidades, como ocorre em métodos exatos.

Assim, para gerar números aleatórios, foram apresentadas várias funções, dentre elas, a função **random()**, que permite produzir números aleatórios reais entre **0** e **1**. Também a função **uniform()** que permite gerar números aleatórios reais, mas que estejam entre uma faixa de valores. Para números aleatórios inteiros, a função **randint()** e por fim, a função **randrange()** que permite gerar números aleatórios entre uma faixa de valores e estabelecendo o salto que deve ser dado ao sortear os números, permitindo assim, gerar por exemplo, apenas números inteiros, números pares, ou números que sejam múltiplos de algum valor.

7.7 Exercícios da Aula

1. Faça um programa em Python que gere e imprima 10 números aleatórios. Utilize laço para isso.

2. Faça um programa que gere 20 números aleatórios e armazene em uma lista, após isso, percorra a lista e imprima os números aleatórios maiores que **0.5**.

3. Escreva um programa que faça um sorteio de números da loteria, para isso, observe a faixa de números do sorteio e a característica dos números.

4. Escreve um programa que imprima 100 números aleatórios inteiros positivos múltiplos de 5.

5. Faça um programa que defina uma lista com tamanho de 100, após isso, em um laço o programa deve gerar números inteiros aleatórios entre 1 e 100, e a cada número produzido preencher a posição correspondente na lista com outro número aleatório, mas neste caso pertencente ao conjunto dos reais. Se uma mesma posição for sorteada duas vezes, o programa deve descartar o novo valor sorteado. O programa deve executar até que todas as posições sejam preenchidas. Para isso, faça uma função que verifique se a lista já está toda preenchida.

Parte II

Parte II - Aplicando a linguagem Python com métodos Heurísticos de Otimização

Ao longo da parte **I** foram introduzidos os conceitos da linguagem Python, apenas o mínimo suficiente para programar as heurísticas que serão implementadas nesta parte do livro. A partir da parte II, o objetivo é aplicar os conceitos aprendidos na área de otimização, contudo, como o objetivo é, de apenas abordar conceitos introdutórios, serão tratadas apenas algumas heurísticas e meta-heurísticas.

Na próxima aula, 8, serão abordados os conceitos iniciais sobre o uso de heurísticas em otimização. Na aula 9, a parte prática é iniciada com algumas heurísticas de busca, de refinamento e de intensificação. As aulas seguintes tratarão de algumas meta-heurísticas como: o **GRASP** (aula 10), a **busca Tabu** (aula 11) e o **Simulated Annealing** (aula 12). Os exemplos práticos implementados, sejam de heurísticas ou meta-heurísticas, se limitaram a um problema clássico de otimização, o problema da mochila múltipla. Pois assim, será mais fácil adaptar o problema aos diferentes exercícios que serão realizados.

AULA 8

Métodos Heurísticos de Otimização

Metas da Aula

1. Compreender a eficiência na aplicação dos métodos heurísticos em problemas de otimização.
2. Entender a diferença dos métodos heurísticos para os métodos exatos.
3. Aprender conceitos introdutórios de problemas de otimização.
4. Preparar o caminho para as próximas aulas.

Ao término desta aula, você será capaz de:

1. Determinar se, para um problema de otimização, será melhor aplicar um método exato ou um método heurístico para a sua resolução.
2. Entender quais as vantagens e desvantagens do uso de métodos heurísticos.

8.1 Porque métodos heurísticos para resolver problemas de otimização?

Na aula 7 já foi falado brevemente sobre a complexidade de resolução dos problemas considerados NP-difícil (GOLDBARG; LUNA, 2005; GONZALEZ, 2007; GOLDBARG; GOLDBARG; LUNA, 2016) com métodos exatos, que são os métodos tradicionais para resolução de problemas de otimização, e grande parte dos problemas de otimização clássicos que são à base dos demais problemas, infelizmente, tem complexidade igual a NP-difícil, como: o **PRM - problema de recobrimento mínimo** (GOLDBARG; LUNA, 2005, p. 420), o **problema de Steiner**, o **problema do caminho mínimo**, o **problema de roteamento**, o **problema de alocação de tarefas**, o **problema do caixeiro viajante** (GLOVER; KOCHENBERGER, 2003; GONZALEZ, 2007), o **problema de programação de produção do sistema de fluxo** (KUMAR; SURESH, 2009, p. 249) entre outros.

O problema do Caixeiro Viajante é ótimo, para exemplificar o grau de complexidade dos problemas NP-difícil. Observe na figura 9, neste exemplo o caixeiro deve passar por 4 cidades, todas as cidade possuem caminhos para as demais cidades. O objetivo é que o caixeiro adote a rota que minimize o seu custo de viagem, reduzindo os quilômetros percorridos, por exemplo.

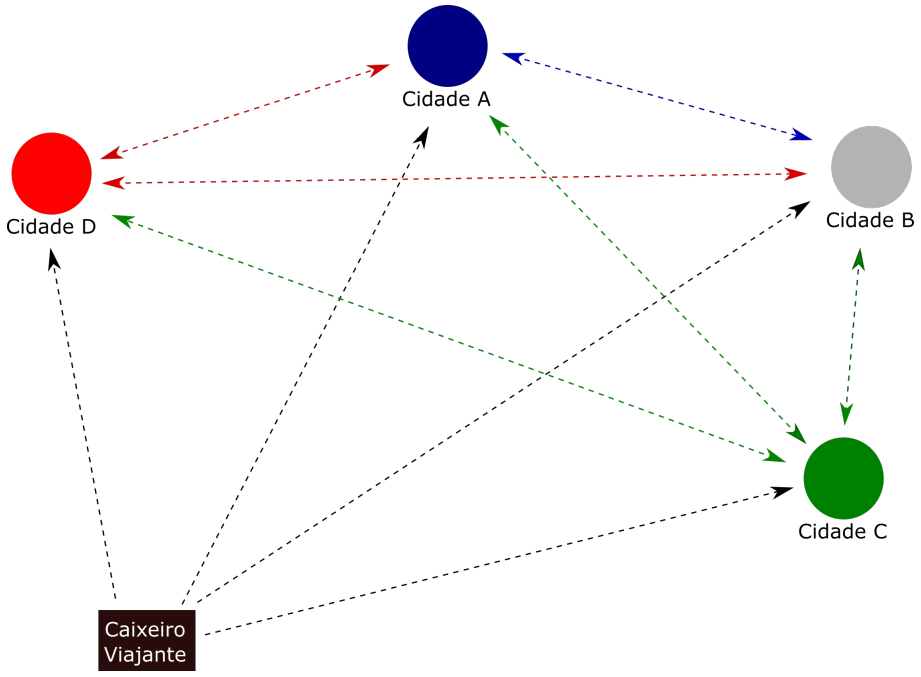

Figura 9 – Exemplo de rota para o Caixeiro Viajante

Este problema tem sua complexidade determinada pelo fatorial do número de cidades ou localidades que devem ser visitadas -1, ou seja, *(n-1)!*. Como no exemplo são 4 localidades, então a sua complexidade é determinada por $(4-1)!$, ou seja, $3 \times 2 \times 1 = 6$. Assim, para saber qual é rota com menor custo, ou seja a rota ótima, deve-se calcular todas as possíveis rotas e selecionar a que resultar em menor custo, o que equivale à menor função objeto (FO), no jargão da otimização. No caso do método exato, a sua busca pretende obter a função objetivo ótima, então, ele examinará todas as possíveis combinações [1]. Até ai, tudo bem! uma pessoa não precisa nem de computador para

[1] Há algoritmos de solução exata que conseguem obter a solução ótima sem examinar todas as combina-

calcular 6 combinações de rotas, mas e se aumentarmos para 5 cidades, então teríamos 4! = 24 combinações possíveis, ainda está fácil, mas aumentando para 6 cidades, teremos 5! = 120, opa! Aumentou consideravelmente, vamos precisar de computador para resolver. Contudo, observe na tabela 9 que ao aumentar para 7 cidades, o número de combinações possíveis aumentou para 720, aumentando para 10 cidades o computador já terá dificuldade para resolver o problema com o método exato. Com valores acima disso, é provável que o computador vai levar anos para resolver, tornando inviável o uso de método exato na resolução.

Tabela 9 – Combinações de rotas para o problema do caixeiro viajante

Nº Localidades		Número de rotas
7	6!	720
10	9!	362.880
20	19!	$(1.216451004 \times 10)^{17}$
30	29!	$(8.841761994 \times 10)^{30}$

Fonte: Os autores

É ai que entram os métodos aproximativos, pois apesar da desvantagem de não garantirem a solução ótima, no geral, com os métodos aproximativos é possível obter soluções próximas da solução ótima, mas com o tempo de processamento muito inferior aos de métodos exatos, mesmo em casos em que o número de variáveis é muito grande (GOLDBARG; GOLDBARG; LUNA, 2016, p. 46).

8.2 Métodos exatos X Métodos heurísticos

Entender a diferença dos métodos exatos para os métodos heurísticos é muito importante, como já mencionado, os métodos exatos atuam analisando todas as combinações possíveis [veja a nota de rodapé], calculando a função objetivo de cada solução, em busca daquela que oferecerá o maior valor, em problemas de maximização, ou o menor valor, em problemas de minimização. A este valor ótimo, denominamos como "ótimo global", seja para um problema de maximização ou de minimização.

Os métodos heurísticos, por sua vez, não avaliam todas as possíveis soluções (combinações), por isso, não há como garantir que um método heurístico irá encontrar a solução que tem o ótimo global para o problema, mas, no geral, um método heurístico consegue obter soluções de boa qualidade, que denominamos como "ótimo local"(GOLDBARG; GOLDBARG; LUNA, 2016, p. 47). Mas, porque ótimo local? Imagine que em um determinado espaço amostral de dados, há vários pontos de máximo, conforme demonstrado na figura 10. Observe que dentre os vários pontos que foram marcados, o que atinge o ponto mais alto foi marcado com a cor vermelha, este é o ótimo global, os demais pontos de máximo foram marcados com a cor azul, estes são os ótimos locais, pois embora sejam pontos em que a solução atingiu valores altos, não se equiparam ao valor máximo do espaço de soluções.

É importante salientar que na figura foi representado um problema de maximização, por isso os pontos de máximo estão na parte superior do gráfico, se fosse um problema de minimização, então os pontos de mínimo seriam destacados na parte inferior do gráfico. Como já mencionado, com um método heurístico não é possível ter certeza de que o ponto de máximo alcançado se trata do ótimo global, pois não há como

ções, como por exemplo, o *branch-and-bound* (ARORA, 2004, p. 516) e o *branch-and-cut* (FARAHANI; HEKMATFAR, 2009, 129), contudo, ainda assim, o número de combinações avaliadas é muito grande.

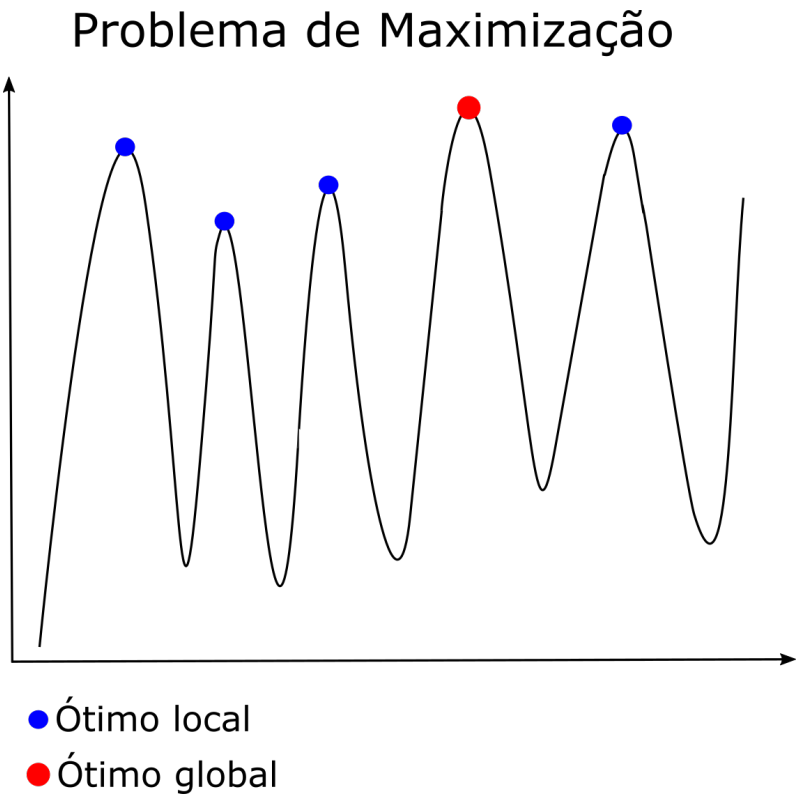

Figura 10 – Ótimo Global x Ótimo local

determinar qual dos pontos de ótimo o método heurístico atingiu [2]. Mas porque? Simplesmente porque no método heurístico as soluções não são exploradas em sua totalidade, como apenas parte é explorada, não há como garantir que o ótimo global foi visitado, neste caso, não será possível determinar que a solução encontrada por uma heurística se trata de um ótimo global, apenas um ótimo local.

É possível resumir então da seguinte forma: Métodos exatos permitem encontrar a solução que atinge o ótimo global de um determinado problema, contudo, na prática só conseguimos aplicar os métodos exatos em problemas de baixo grau de complexidade, que em geral, está associado ao número de variáveis que fazem parte do problema. Já os métodos heurísticos são aproximativos, pois não é possível determinar se o ótimo global foi atingido, apenas um ótimo local, porém, podem ser aplicados em problemas de alta complexidade, ou seja, com alto número de variáveis associadas ao problema, em um tempo razoável de processamento pelo computador.

De forma que, os métodos heurísticos estão ganhando muito espaço neste ramo, pois tem se mostrado uma alternativa válida para a aplicação da otimização no mundo real, visto que, na prática, é muito comum que os problemas tenham um alto grau de complexidade, sendo assim inviável o uso dos métodos exatos (GOLDBARG; GOLDBARG; LUNA, 2016, p. 47).

[2] Exceto, nos casos em que foi possível obter o ótimo global pelo método exato, assim, será possível comparar com o resultado da heurística.

8.3 Como as heurísticas funcionam

Mas então, como funcionam as heurísticas? Como é possível que este método avalie várias soluções, mas desprezando outras? Há várias estratégias de funcionamento para uma heurística, inclusive, há heurísticas que combinam duas ou mais estratégias, mas basicamente as heurísticas estudadas neste livro utilizam quatro técnicas básicas: a aleatoriedade, a gulosidade, o refinamento e a intensificação.

A aleatoriedade permite que a heurística percorra o espaço amostral de soluções de forma casual, sempre buscando novas soluções sem um caminho especifico, esta característica, no geral, favorece a obtenção de boas soluções. A gulosidade assume que as combinações que produzem os valores mais altos tem maiores chances de produzirem as melhores soluções, já o refinamento é uma estratégia que, geralmente está associada a uma busca local. Imagine o seguinte, suponha que ao selecionar uma solução de forma aleatória, esta não seja a que trará o ótimo global, mas a mesma está muito próxima dá melhor solução do problema, assim, ao adotar apenas o componente da aleatoriedade, é possível que a heurística passe muito perto de obter a melhor solução, mas sem sucesso, contudo, se a busca for refinada de modo a avaliar também os vizinhos de cada solução encontrada, a probabilidade de encontrar a solução ótima aumenta consideravelmente. Por fim, a intensificação atua de forma similar, ao encontrar uma região promissora no espaço de soluções, a heurística intensificará as buscas naquela região, na expectativa de encontrar o ótimo global.

8.4 Problema da Mochila

Não é o objetivo deste livro abordar exaustivamente os conceitos sobre otimização, métodos exatos ou heurísticos, pois espera-se que o leitor já tenha alguma bagagem teórica para que possa usufruir da bagagem prática que pretende-se abordar. Assim, neste momento o caminho será traçado para as próximas aulas, em que, serão aplicados, na prática o uso da linguagem Python em otimização. Para isso, o problema da mochila será adotado como padrão para as heurísticas e meta-heurísticas que serão implementadas.

O problema da mochila (em inglês: *knapsack problem*) é simples de entender, e além disso, caracteriza-se com um grande número de outros problemas de programação (GOLDBARG; GOLDBARG; LUNA, 2016, p. 8). Para entender o problema, imagine que está para fazer uma aventura de escalada em uma montanha, de forma que, você precisa levar os itens básicos para a sua jornada, mas há uma limitação da capacidade da mochila, para que você não leve muito peso, pois quanto mais peso levar, mais cansativa será a subida. Assim, você pegou os itens de interesse e atribuiu a cada um, certo grau de importância, para que, com base em sua importância e seu peso, determinar quais itens irão objetivando maximizar o valor que será levado na mochila, dada a limitação da capacidade.

Simples, não é? Você tem uma mochila, mas esta tem uma limitação na capacidade de carga, assim, precisa escolher quais itens irá levar, para isso, determinamos para cada item o seu peso e o "valor", ou grau de importância, para que seja possível escolher os itens visando maximizar o valor considerando a capacidade da mochila. Bem, para que o problema não seja muito trivial, a versão adotada é um pouco mais complexa, em que, ao invés de se limitar a uma mochila apenas, não haverá limite para a quantidade de mochilas, ou seja, m mochilas e n itens. Esta versão do problema é denominada como: **problema da mochila múltipla (PMM)** (GOLDBARG; GOLDBARG; LUNA, 2016, p. 11), em inglês: *multiple knapsack problem* (MKP). Assim, as heurísticas serão imple-

mentadas com a capacidade de determinar quais objetos serão alocados, considerando que há várias mochilas.

8.5 Estrutura de Dados

A definição da estrutura de dados que será adotada na construção da heurística pode fazer toda a diferença no desempenho. O problema da mochila pode ajudar a entender isso. Antes de continuar, é preciso definir a estrutura para o problema já mencionado. Para facilitar o entendimento de como definir a estrutura, pode-se associar com a forma como pretende-se armazenar os dados que serão manipulados ao longo da execução da heurística.

Para o problema da mochila múltipla, deve-se armazenar vários elementos, os itens ou objetos da mochila, as mochilas, o peso dos itens e o valor dos itens. Contudo, estes elementos serão utilizados apenas para consulta, de forma que, serão armazenados apenas uma vez no inicio do processo, neste caso, não há grande preocupação com a estrutura. Durante a execução da heurística será necessário manipular constantemente as soluções encontradas. Para registrar a solução, é preciso armazenar a mochila e os itens alocados nela. Assim, ao pensar em como deve ser essa estrutura é natural que tenhamos a intuição de pensar que ela deve ter a hierarquia **Mochila » Itens**, conforme a figura 11.

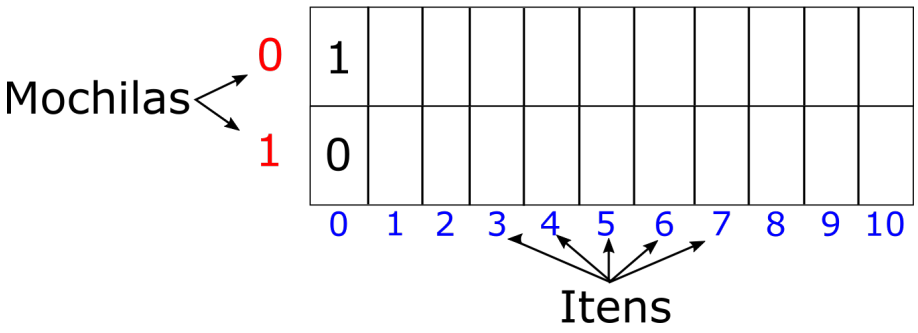

Figura 11 – Primeira proposta de Estrutura de Dados para o problema da mochila múltipla (PMM)

Observe na figura 11 que esta estrutura permite armazenar 2 mochilas e para cada mochila, até 11 itens. É muito comum neste tipo de problema utilizarmos **0** e **1** nos registros. Exemplo, se o **item 0** for armazenado na **mochila 0**, então armazenamos **1** na posição (0, 0) da matriz e **0** na posição (1, 0), conforme o exemplo apresentado na figura 11. Ou seja, estamos dizendo que o **item 0** foi armazenado na **mochila 0**, e portanto, como um mesmo item não pode ser armazenado em duas mochilas, ele não foi armazenado na **mochila 1**. Assim, dada essa restrição do item não poder ser armazenado em mais de uma mochila, se utilizarmos essa estrutura será necessário tratar esta questão ao realizar as buscas por soluções, pois caso contrário poderão ocorrer conflitos como o representado na figura 12.

A figura 12 mostra uma situação em que o **item 1**, foi aleatoriamente selecionado para as duas mochilas, isso é uma quebra da restrição, pois um mesmo item não pode ser armazenado em duas mochilas. Bem, o programador pode então tratar essa questão da seguinte forma, sempre que a heurística selecionar um item, verificar se aquela

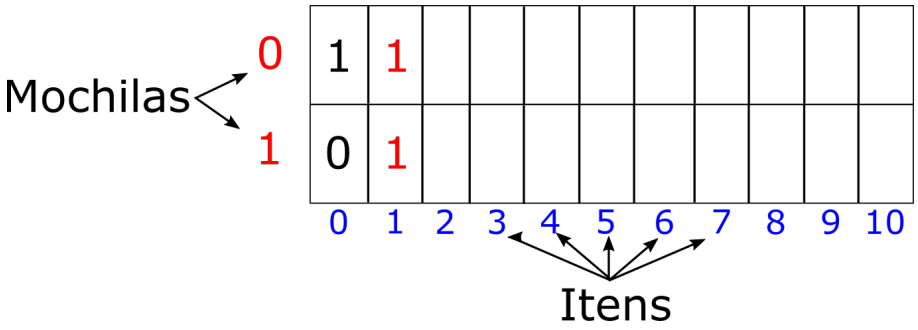

Figura 12 – Primeira proposta de Estrutura de Dados para o problema da mochila (PMM) com conflito

seleção não causa uma quebra de restrição, e se quebrar, fazer uma nova busca por uma seleção que não cause a restrição, contudo, fazer esse tratamento irá implicar em perda de desempenho no processamento. Como já mencionado, a estrutura de dados pode fazer diferença no desempenho da heurística, então uma segunda possibilidade é que o programador tente adequar a estrutura de forma a minimizar a necessidade de tratar as restrições, ou seja, estruturar os dados, de forma que, o tratamento da restrição seja natural. A figura 13 mostra uma adequação da estrutura com este objetivo.

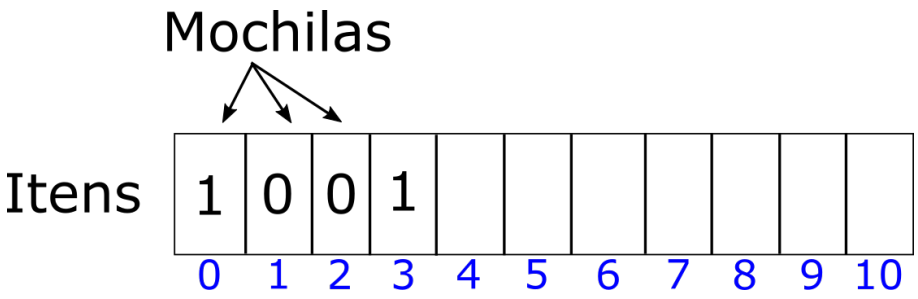

Figura 13 – Segunda proposta de Estrutura de Dados para o problema da mochila múltipla (PMM)

Agora sim! A adequação realizada na proposta apresentada na figura 13 torna os itens fixos e quem passa a ser armazenado é a mochila, assim, fica impossível que um item seja armazenado em mais de uma mochila, concorda? Veja que ótima solução, pois, foi possível resolver uma restrição sem nenhum tratamento de programação, como condições ou laços, apenas ajustando a estrutura de dados, assim, o desempenho da heurística será melhor, pois exigirá um número menor de instruções de verificação e busca.

De fato, a estrutura de dados adotada pode ser fundamental no ganho de desempenho, contudo, não é uma tarefa trivial, pois cada problema demanda uma estratégia diferente e alguns problemas nem mesmo permitem variações da estrutura, mas é fundamental que o programador tenha em mente duas coisas:

1. Primeiro, é de suma importância fazer o exercício de buscar variações da estrutura de dados para o problema com o objetivo de reduzir instruções de validação e busca.

2. Segundo, este trabalho deve ser realizado antes de iniciar a construção da heurística, pois o esforço de fazer o ajuste após a construção pode tornar este processo inviável.

Ao longo das próximas aulas, em que serão construidas as heurísticas e metaheurísticas para o problema da mochila múltipla (PMM), a estrutura de dados apresentada na figura 13 será adotada como padrão.

8.6 Resumo da Aula

Esta aula permitiu entender um pouco sobre a importância do uso de heurísticas no estudo dos problemas de otimização. Ficou claro que, embora os métodos exatos possibilitem obter a melhor solução, ou o ótimo global, na prática, dado o crescimento exponencial das combinações que devem ser avaliadas, não há sucesso em aplicar estes métodos em muitos problemas práticos. Assim, os métodos heurísticos são uma valiosa alternativa, pois apesar de que, não há garantia de obtenção da solução que garante o ótimo global, no geral, os métodos heurísticos são eficientes em obter boas soluções com ótimos locais com um esforço computacional considerado razoável (GOLDBARG; GOLDBARG; LUNA, 2016, p. 47).

Outro ponto importante foi a definição do problema que iremos adotar a partir da próxima aula, o problema da mochila múltipla (PMM), com m mochilas e n itens, pois tal problema além de simples de entender, tem uma relação forte com outros vários problemas presentes no mundo real. Seu uso para o aprendizado será de grande valor.

O último tópico da aula foi a discussão acerca da estrutura de dados. Vimos que a estrutura pode ser fundamental no ganho de desempenho da heurística e após 2 opções de estrutura serem apresentadas, chegou-se a uma proposta em que a estrutura para o **PMM**, elimina a necessidade de validar a restrição de alocação de um mesmo item em mochilas distintas, o que trará ganho de desempenho no processamento das heurísticas e meta-heurísticas.

AULA 9

Heurísticas

Metas da Aula

1. Implementar as principais heurísticas para o problema da mochila múltipla (PMM).
2. Entender as diferenças e particularidades das heurísticas implementadas.
3. Aprender o uso da linguagem de programação Python na implementação de heurísticas.

Ao término desta aula, você será capaz de:

1. Implementar heurística em linguagem Python.
2. Entender quais as vantagens e desvantagens no uso das heurísticas implementadas.
3. Determinar qual heurística melhor se enquadra em um determinado problema pela implementação e teste.
4. Implementar os principais elementos presentes em problemas de otimização, como: o cálculo da função objetivo e a leitura dos dados.

9.1 Um pouco sobre Heurísticas

Boa parte dos autores entende que uma heurística, em problemas de otimização, deve garantir ao menos uma solução viável como resultado, mas sem garantir a qualidade da solução, ou seja, sem garantir que será obtida uma solução com ótimo global ou um ótimo local próximo do ótimo global. Goldbarg, Goldbarg e Luna (2016) trazem a seguinte definição para heurística:

> Uma heurística é uma técnica computacional que alcança sempre uma solução viável para um dado problema de otimização, utilizando um esforço computacional considerado razoável. (GOLDBARG; GOLDBARG; LUNA, 2016, p. 47)

Quando o esforço computacional é considerado razoável? Isso é bem relativo, pode variar de problema para problema. No geral, as heurísticas serão computacionalmente mais eficientes do que os métodos exatos para muitos problemas, em especial os de grande complexidade. Contudo, mesmo uma heurística pode vir a não ter a capacidade de disponibilizar a resposta necessária em tempo razoável. Exemplo, imagine um problema em que a demanda pela resposta é em tempo real, por se tratar de uma linha de produção. E, após implementar a heurística, o melhor tempo de execução obtido foi de 10 minutos, neste caso, a heurística não será satisfatória, pois, no caso em questão, 10 minutos não atenderia a demanda pela resposta em tempo real na linha de produção, mesmo que os 10 minutos da heurística sejam um avanço em relação ao método exato, que talvez leve meses ou anos para disponibilizar a resposta.

Da mesma forma, é possível que para alguns problemas, o tempo que uma heurística leve, de 2 ou 3 dias para disponibilizar uma resposta, possa ser considerado como um tempo razoável, pois se o problema não demanda a resposta em tempo real, como seria o caso, por exemplo, de um planejamento anual de uma empresa, então, não seria, necessariamente ruim, levar esse longo tempo para processar. Assim, o problema é determinante para estabelecer essa condição.

Historicamente as primeiras heurísticas foram desenvolvidas para atender às necessidades especificas de um determinado problema, ou seja, no geral, não eram aplicáveis a outras classes de problemas (GOLDBARG; GOLDBARG; LUNA, 2016, p. 48). Contudo, não levou muito tempo até surgirem as primeiras heurísticas "genéricas", por assim dizer, denominadas como meta-heurísticas, estas, possibilitam o seu uso em variados problemas com pouca ou nenhuma adaptação.

9.2 Heurísticas Construtivas

As heurísticas construtivas tem o objetivo de construir uma solução, elemento por elemento. O elemento escolhido vai variar de acordo com a função objetivo, de forma que, no geral o algoritmo base deverá ser adaptado para cada problema. A escolha pode ser aleatória ou seguir algum critério pré-estabelecido, como a gulosidade. Quando a seleção é aleatória, geralmente a implementação é muito simples, contudo, a solução gerada não costuma ser de qualidade. Geralmente, tem-se o objetivo de obter uma solução viável, mesmo que ruim. Caso a solução não seja viável, algumas estratégias podem ser adotadas, como: buscar uma nova solução ou penalizar a solução encontrada. Veja a seguir o algoritmo base para a heurística construtiva.

```
1 S ← ∅
2 DEFINIR (conjunto C de elementos candidatos)
3 ENQUANTO (C ≠ ∅) FACA
```

```
4     SELECIONAR (um elemento e∈C)
5     S ← S∪e
6     C ← C−e
7  FIM-ENQUANTO
8  RETORNAR (S)
```

A heurística construtiva é muito simples, como pode-se ver no código-base. Primeiro foi declarada a variável responsável por armazenar a solução, inicialmente vazia, conforme a linha 1, na sequência define-se o conjunto de elementos candidatos à compor a solução, na linha 2. Na linha 3, um laço determina que iterações devem ser executadas enquanto houverem elementos candidatos a fazerem parte da solução. Na linha 4, um elemento candidato será selecionado em cada iteração. Na linha 5 a solução recebe o elemento que foi selecionado e na linha 6 o elemento que foi adicionado à solução é removido do conjunto de elementos candidatos. Ao final do procedimento a solução retorna ao método que a invocou, conforme a linha 8.

9.3 Heurística Construtiva Gulosa (*Greedy*)

As heurísticas construtivas gulosas ou *greedy* tem o princípio básico da seleção da melhor solução atual, por exemplo: selecionar o menor valor, selecionar o caminho mais curto ou selecionar o maior peso, além disso, ao tomar a decisão de seguir o caminho que proporciona a melhor solução, não haverá mais retorno ao passo anterior (GONZALEZ, 2007, p. 48). Para isso, geralmente os elementos são ordenados de acordo com o objetivo, para que, a cada passo seja inserido o elemento que produza a melhor solução. Em relação ao código-base, basta adaptar o algoritmo já apresentado para a heurística construtiva, no passo SELECIONAR.

9.4 Implementando a primeira heurística em Python para o PMM

Para implementar a primeira heurística, será necessário antes, executar uma série de passos até ter o programa mínimo necessário para executá-la. O primeiro passo é definir os dados que serão utilizados para o problema. O segundo passo é construir o programa capaz de ler esses dados.

9.4.1 Definindo os dados para o PMM

Primeiro é necessário definir os dados do problema da mochila múltipla. Como estabelecido na aula 8, as heurísticas e meta-heurísticas serão implementadas para o PMM considerando m mochilas e n objetos. Inicialmente, um arquivo com poucos dados facilitará a analise da eficiência da heurística, de forma que, a seguir foi disponibilizado um conjunto de dados considerando apenas 3 mochilas e 20 itens. Para construir o arquivo de testes, repita os passos já executados na aula 6, mas agora com os novos dados, faça:

Passos

1. Copie os dados disponibilizados a seguir;
2. Cole em um editor de textos plano, como o **notepad**, **notepad++**, ou outro de sua preferência;
3. Salve o arquivo com o nome: **dadosMochila1.txt** na pasta criada na aula 6, **C:\AulaPython**, pois vamos referenciar ela em nossos testes.

Dados para serem salvos no arquivo

```
20 3
644 815 723 111 991 900 492 511 291 518 243 473 144 720 432 834 749 728 318 838
15 4 47 49 3 48 13 80 60 63 86 80 27 18 12 34 37 23 13 16
168 304 239
```

Fonte: Dados fictícios

9.4.2 Lendo os dados

De posse dos dados salvos, basta utilizar o mesmo código empregado na importação da aula 6, contudo, agora é um bom momento para estruturar melhor o código-fonte e separá-lo em funções e arquivos, conforme estudado na aula 5. Desta forma, para criar o arquivo de código-fonte que será responsável pela importação dos dados, siga os passos:

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Cole o código em um editor de textos plano, como o **notepad**, **notepad++**, ou outro de sua preferência;
3. Salve o arquivo com o nome: **leDados.py** na pasta **C:\AulaPython**.

```
1  def leDados():
2      #inicializando as variaveis
3      numObj = numMoc = 0
4      valObj = []
5      pesObj = []
6      capMoc = []
7      #lendo os dados
8      f = open(r"C:\AulaPython\dadosMochila1.txt", "r")
9      #lendo a primeira linha
10     linha = f.readline()
11     valores = linha.split()
12     #lendo o numero de objetos
13     numObj = int(valores[0])
14     #lendo o numero de mochilas
15     numMoc = int(valores[1])
16
17     #lendo a segunda linha
```

```
18      linha = f.readline()
19      valores = linha.split()
20      #lendo os valores dos objetos
21      for val in valores:
22          valObj.append(int(val))
23
24      #lendo a terceira linha
25      linha = f.readline()
26      valores = linha.split()
27      #lendo os pesos dos objetos
28      for val in valores:
29          pesObj.append(int(val))
30
31      #lendo a quarta linha
32      linha = f.readline()
33      valores = linha.split()
34      #lendo a capacidade das mochilas
35      for val in valores:
36          capMoc.append(int(val))
37
38      f.close()
39      return numObj, numMoc, valObj, pesObj, capMoc
```

O código-fonte disponibilizado inclui uma função que, conforme a linha 1, não recebe nenhum argumento e conforme a linha 39, retorna 5 argumentos, pois como o objetivo da função é de apenas ler os dados relativos ao PMM, então não foram necessários dados de entrada, contudo é necessário que a função retorne várias informações do problema, número de objetos, número de mochilas, os valores dos objetos, os pesos dos objetos e a capacidade das mochilas. O arquivo de dados, na linha 8, poderia ser um dado de entrada, mas optou-se por fazer a configuração desta informação nesta função.

Note que, na linha 8 o arquivo é aberto e na linha 38, o arquivo é fechado, assim foi tomado o devido cuidado de fechar o arquivo após o uso. Entre as linhas 10 e 15 foram lidos os valores de número de objetos e número de mochilas. Após ler estes valores, foram necessários laços para ler os demais valores, pois irão variar conforme o número de objetos e número de mochilas, exemplo, se o número de mochilas é 5, então devem ser lidos os valores de capacidade de mochila para as 5 mochilas, conforme as linhas 32 à 36.

9.4.3 Implementando a Heurística Construtiva Aleatória

Com os dados definidos e a função de importação implementada, é possível implementar a heurística, contudo, para testá-la, ainda será necessário desenvolver um programa principal que faça o papel de invocar as funções implementadas. A primeira heurística que será implementada é a **Construtiva Aleatória**, dada a sua simplicidade. Execute os passos:

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Cole o código em um editor de textos plano, como o **notepad**, **notepad++**, ou outro de sua preferência;
3. Salve o arquivo com o nome: **heuConAle.py** na pasta **C:\AulaPython**.

```
1  import random as rand
2  #gerando a semente
3  rand.seed()
4
5  def heuConstrutivaAleatoria(numObj, numMoc):
6      #inicializando o vetor de solucao com
7      #o tamanho do numero de objetos
8      sol = [0] * numObj
9      for i in range(numObj):
10         #para cada objeto e sorteado em
11         #qual mochila ele sera alocado
12         sol[i] = rand.randint(0, numMoc)
13
14     return sol
```

A primeira linha do código-fonte é responsável pela importação da biblioteca **random** que será utilizada para gerar os números aleatórios. A linha 3 é responsável por gerar a semente, isso é necessário, pois caso contrário, a cada execução do programa os números gerados serão iguais, ou seja, não haverá aleatoriedade. Entre as linhas 5 e 14 foi codificada a função *heuConstrutivaAleatoria(numObj, numMoc)* que será responsável por gerar uma solução para o PMM de forma aleatória. Note que a função recebe os argumentos *numObj*, referente ao número de objetos e *numMoc*, referente ao número de mochilas, pois esses dois argumentos são suficientes para produzir a solução aleatória para o problema.

Na linha 8 foi inicializado um vetor para armazenar a solução com o tamanho do número de objetos, seguindo a estrutura que foi definida na aula 8, veja que na linha 9 foi realizado um laço que vai de **0** até o número de objetos, pois, o objetivo é, para cada objeto, sortear em qual mochila ele será alocado, conforme realizado na linha 12. Assim, será garantida a restrição de que um objeto não pode ser alocado ao mesmo tempo em mais de uma mochila, sem a necessidade de implementar alguma validação para isso. Por fim, a linha 14 é responsável por retornar a solução gerada para o código que invocou a função.

9.4.4 Construindo o programa principal

O arquivo de dados do problema já está pronto, a função responsável pela leitura dos dados também, da mesma forma, a primeira heurística aleatória, então é preciso construir a aplicação principal que será responsável por invocar essas funções e imprimir o resultado da heurística que foi implementada. Similar ao que foi feito antes, execute os seguintes passos:

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Cole o código em um editor de textos plano de sua preferência;
3. Salve o arquivo com o nome: **prgMain.py** na pasta **C:\AulaPython**.

```
1  import leDados as LD
2  import heuConAle as HCA
3
4  numObjetos = numMochilas = 0
5  vetValoresObjetos = []
6  vetPesosObjetos = []
7  vetCapacidadeMochilas = []
8  numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas = LD.leDados()
9
10 #obtendo uma solucao construtiva aleatoria
11 sol = HCA.heuConstrutivaAleatoria(numObjetos, numMochilas)
12
13 print "Solucao: ", sol
```

Note que nas linhas 1 e 2 foi realizada a importação dos dois arquivos em que salvamos a função responsável pela leitura dos dados e a função responsável pela heurística construtiva aleatória, pois assim, será possível invocar os métodos presentes nestes arquivos, conforme foi realizado a invocação da função *leDados()* na linha 8 e a invocação do método *heuConstrutivaAleatoria()* na linha 11. Observe também que entre as linhas 4 e 7 foram inicializadas as variáveis e os vetores que irão receber os dados lidos pela função *leDados()*, conforme a linha 8. Na linha 11 também foi declarada a variável *sol* que é utilizada para receber a solução da função *heuConstrutivaAleatoria()* e imprimir este resultado na linha 13.

A partir deste momento é possível fazer o teste da heurística, pois o programa mínimo necessário para executá-la já está concluído, para isso, pode-se utilizar um interpretador Python. Faça o teste executando várias vezes e notará que receberá uma resposta diferente a cada execução, pois, as soluções são sorteadas aleatoriamente. Veja a seguir como executar o código em modo interativo.

```
1  >>> runfile('C:/AulaPython/prgMain.py', wdir='C:/AulaPython')
2  Solucao:  [1, 1, 0, 1, 2, 1, 0, 2, 3, 3, 0, 2, 1, 3, 1, 3, 0, 2, 3, 3]
```

Observe que ao executar o programa **prgMain.py** na linha 1, foi impressa a solução obtida aleatoriamente na linha 2, assim, se você fizer o teste executando várias vezes, terá vários resultados diferentes. Caso tenha salvo os arquivos em um diretório diferente, basta alterar o diretório no código da linha 1.

Observe também, na linha 2, que entre os 20 itens, alguns foram armazenados na mochila **1**, alguns na mochila **2**, alguns na mochila **3** e alguns não foram alocados em mochila, é o caso dos itens cujo valor é **0**. É de se esperar que nem todos sejam armazenados, pois as mochilas não podem comportar todos os itens, caso pudessem não seria necessário otimizar. Até o momento, o nosso programa apenas exibe a solução, ou seja, os itens e a sua alocação ou não em uma mochila, mas qual é a função objetivo para a solução sorteada? O cálculo da função objetivo ainda não foi feito, por isso, este é o próximo passo. Faça:

Passos

1. Copie o PRIMEIRO código-fonte disponibilizado a seguir;
2. Cole o código em um editor de textos plano de sua preferência;
3. Salve o arquivo com o nome: **calcFO.py** na pasta **C:\AulaPython**;
4. Copie o SEGUNDO código-fonte disponibilizado;
5. Substitua pelo código do arquivo **prgMain.py** e salve.

PRIMEIRO código-fonte:

```
1  def calcFO(solucao, valObj, numObj):
2      fo = 0
3      #executa um laco percorrendo os objetos
4      for i in range(numObj):
5          #calcula a FO dos objetos selecionados
6          #apenas se ele foi alocado em mochila
7          if solucao[i] != 0:
8              fo += valObj[i]
9
10     return fo
```

SEGUNDO código-fonte:

```
1  import leDados as LD
2  import heuConAle as HCA
3  import calcFO as CF
4
5  numObjetos = numMochilas = 0
6  vetValoresObjetos = []
7  vetPesosObjetos = []
8  vetCapacidadeMochilas = []
9  numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas = LD.leDados()
10
11 #obtendo uma solucao construtiva aleatoria
12 sol = HCA.heuConstrutivaAleatoria(numObjetos, numMochilas)
13
14 #calculando a FO da solucao obtida
15 FO = CF.calcFO(sol, vetValoresObjetos, numObjetos)
16
17 print "Solucao: ", sol
18 print "FO: ", FO
```

O primeiro código-fonte disponibilizado é o método que calcula a função objetivo para o problema, observe que a função *calcFO()* recebe três argumentos, o primeiro, *solucao* refere-se ao vetor que terá os itens com as mochilas em que foram alocados, o segundo, *valObj* é outro vetor com os valores para cada objeto, e o terceiro, *numObj* é o número de objetos. Com estes três argumentos de entrada é possível calcular a *FO* [1]. Na linha 2 foi inicializada a variável que será responsável por armazenar o valor da *FO*, entre

[1] A abreviação *FO* refere-se ao valor da função objetivo.

as linhas 4 e 8, foi realizado um laço pelos itens somando os seus respectivos valores, sempre que o mesmo foi alocado em alguma mochila, esta validação foi realizada com a condição **if** presente na linha 7. Por fim, na linha 10, o valor contabilizado para a *FO* é retornado para o método que a invocou.

O segundo código-fonte disponibilizado trata-se do código do programa principal ajustado para invocar a função *calcFO()*, observe que na linha 3 foi adicionado um **import** para o arquivo com o código-fonte de *calcFO()*, na linha 15 foi adicionada a invocação para a função que calcula a *FO*, por fim, a linha 18 imprime o valor da função objetivo, as demais linhas de código não sofreram alterações. Agora, teste novamente conforme a seguir:

```
1 >>> runfile('C:/AulaPython/prgMain.py', wdir='C:/AulaPython')
2 Solucao:  [2, 0, 0, 1, 2, 3, 1, 2, 2, 0, 1, 2, 1, 3, 2, 3, 2, 1, 2, 1]
3 FO:  9419
```

Observou que agora foi impresso também o valor da função objetivo, conforme a linha 3? Assim, é possível avaliar se a *FO* melhorou ou não comparando com outros resultados. Além disso, o método do cálculo da *FO* poderá ser reaproveitado na implementação das meta-heurísticas.

9.5 Penalização de soluções inviáveis

Até o momento, o programa desenvolvido não verifica se a solução é viável ou não, ou seja, se os itens alocados em mochilas realmente cabem nas mochilas. Existem duas questões relacionadas à esta validação, a primeira é: em que ponto do programa essa verificação será realizada? A segunda questão é: ao encontrar uma solução inviável, vamos descartá-la ou aceitá-la? Uma decisão depende da outra, se pretende-se descartar as soluções inviáveis, então, é provável que o ideal seja tratar a validação na função que gera a solução aleatória, neste caso, na função *heuConstrutivaAleatoria()*, pois, desta forma, caso a função não seja viável, pode-se tomar a ação de gerar uma nova solução até que obtenha-se uma solução viável. Contudo, isso pode ser um problema, pois não há como determinar quantas soluções serão necessárias até se obter uma solução viável e o desempenho do programa pode cair muito.

Uma boa alternativa é sempre aceitar a solução, mesmo ela sendo inviável, pois assim, o método sempre retornará uma solução com um tempo constante de execução. Contudo, não é desejável obter uma solução inviável como resultado para o problema, não é mesmo? Como proceder então? Pode-se penalizar a solução inviável, tornando ela tão ruim que ao escolher uma nova solução viável mesmo tendo uma *FO* com "baixo valor", esta será "melhor"que a solução inviável e consequentemente, irá substituí-la. Veja um exemplo para que este conceito fique mais claro. A tabela 10 apresenta as possíveis combinações para o problema da mochila (PM), sendo **1** mochila e **3** itens. Os valores dos itens são: {**10, 7, 6**}, respectivamente, e os pesos são: {**5, 3, 4**}, respectivamente, por fim, a capacidade da mochila é: **8**.

Ao analisar a tabela 10, percebe-se que dentre as possibilidades de combinações, duas violam a restrição de capacidade de carga, sendo que a primeira possui a melhor *FO* dentre todas, no caso 23. Então imagine, se o método heurístico simplesmente aceitar todas as soluções, e a primeira solução deste exemplo for sorteada, nenhuma outra seria considerada a "melhor"após o sorteio desta, há não ser que, após verificar que esta solução viola a restrição, o programa penalizar ela ao ponto em que se torne uma solução tão ruim que mesmo a solução que possui a "pior"*FO*, supere o valor desta. Uma forma de penalizar, poderia ser ajustando o valor da *FO* de soluções que violam

Tabela 10 – Combinações para o PM com 3 itens e 1 mochila

Itens			Capacidade Necessária	Carga Excedida	FO
1	2	3			
1	1	1	12	4	23
1	0	1	9	1	16
1	1	0	8	-	17
1	0	0	5	-	10
0	1	1	7	-	13
0	1	0	3	-	7
0	0	1	4	-	6
0	0	0	0	-	0

Fonte: Os autores

a restrição, por exemplo, pela expressão: $Fo = Fo - 100 * (totPes - capMoc)$, em que *totPes* é o somatório do peso dos itens e *capMoc* é a capacidade da mochila.

Tabela 11 – Combinações para o PM com exemplo de penalização

Itens			Capacidade Necessária	Carga Excedida	FO Penalizada
1	2	3			
1	1	1	12	4	-377
1	0	1	9	1	-84
1	1	0	8	-	17
1	0	0	5	-	10
0	1	1	7	-	13
0	1	0	3	-	7
0	0	1	4	-	6
0	0	0	0	-	0

Fonte: Os autores

Conforme pode-se verificar na tabela 11, após penalizar as soluções que violam a restrição de capacidade da mochila, elas ficaram com um valor de *FO* tão "ruim"que mesmo a "pior"solução viável tem a *FO* "melhor". Assim, em uma busca pela melhor solução, a menos que sejam sorteadas apenas soluções inviáveis, dificilmente o programa irá trazer como resultado uma solução inviável. Uma observação importante, é que o método de penalização vai sempre variar de acordo com o problema. Talvez para um outro problema, penalizar com o valor 100 não seja suficiente, ou ainda, pode ser que ao invés de subtrair, seja necessário somar (em um problema de minimização). Assim, cada caso deve ser avaliado para a definição de como proceder com a penalização. Inclusive, em um problema em que as soluções viáveis são raras, pode ser interessante penalizar sem inviabilizar totalmente a possível escolha da solução inviável, pois em alguns casos, a seleção de uma solução inviável, não seria tão prejudicial ao problema.

Caso opte por fazer a penalização da solução inviável, então o melhor ponto para fazer esta validação é a função responsável pelo cálculo da *FO*, ou seja, para o exemplo deste livro, será então o método *calcFO()*. Para ajustar o calculo da função objetivo de forma que seja feita a penalização, siga os passos:

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Abra o arquivo **calcFO.py** da pasta **C:\AulaPython** e substitua o código-fonte deste arquivo pelo código-fonte copiado na área de transferência.
3. Salve o arquivo **calcFO.py**.

```
1  ALFA = 1000
2
3  def calcFO(solucao, numObj, numMoc, valObj, vetPesObj, vetCapMoc):
4      fo = 0
5      vetPes = [0] * numMoc
6
7      for i in range(numObj):
8          #calcula a FO dos objetos selecionados
9          if solucao[i] != 0:
10             fo += valObj[i]
11             vetPes[solucao[i]-1] += vetPesObj[i]
12
13     for j in range(numMoc):
14         #verifica se a capacidade da mochila foi excedida
15         if vetPes[j] > vetCapMoc[j]:
16             #se verdadeiro penaliza a FO
17             fo -= ALFA * (vetPes[j] - vetCapMoc[j])
18
19     return fo
```

O primeiro ajuste feito na função *calcFO()* foi a inclusão da constante *ALFA* que fará o trabalho de penalizar muito ou pouco a *FO*, conforme a necessidade. O segundo ajuste foi nos argumentos de entrada da função, note que agora são 6 argumentos e não 3 como antes, pois para validar se a solução é inviável ou não, serão necessárias novas informações, como o peso dos objetos, a capacidade das mochilas e o número de mochilas. Foi declarado um vetor para armazenar o peso alocado em cada mochila na linha 5, foi também adicionada a linha 11 que é responsável por somar o peso alocado em cada mochila e por fim, foram adicionadas as linhas 13 a 17 que são responsáveis pela validação da solução, e em caso de violação de restrição, a penalização na linha 17. Como foi alterado o número de argumentos de entrada da função, então será necessário ajustar também o programa principal **prgMain.py**, conforme o código a seguir.

```
1  import leDados as LD
2  import heuConAle as HCA
3  import calcFO as CF
4
5  numObjetos = numMochilas = 0
6  vetValoresObjetos = []
7  vetPesosObjetos = []
8  vetCapacidadeMochilas = []
9  numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas = LD.leDados()
10
11 #obtendo uma solucao construtiva aleatoria
12 sol = HCA.heuConstrutivaAleatoria(numObjetos, numMochilas)
```

```
13
14 #calculando a FO da solucao obtida
15 FO = CF.calcFO(sol, numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas)
16
17 print "Solucao: ", sol
18 print "FO: ", FO
```

Observe que apenas a linha 15 sofreu alterações em função da mudança dos argumentos de entrada da função *calcFO()*.

9.6 Implementando a heurística Construtiva Gulosa para o PMM

O próximo passo é a implementação da heurística construtiva gulosa. Neste caso, a estratégia será fazer a ordenação decrescente dos valores dos objetos, assim, a cada iteração o candidato com maior valor ser obtido, por isso esta heurística é denominada como "gulosa", pois assume a estratégia de obter o maior valor a cada passo. Para testar a heurística gulosa para o PMM, realize os passos a seguir:

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Cole o código em um editor de textos plano de sua preferência;
3. Salve o arquivo com o nome: **heuConGul.py** na pasta **C:\AulaPython**.

```
1  import numpy as np
2
3  def heuConstrutivaGulosa(numObj, numMoc, vetValObj, vetPesObj, vetCapMoc):
4      sol = [0] * numObj
5      #este vetor e utilizado para armazenar o peso alocado em cada mochila
6      vetPes = [0] * numMoc
7
8      #obtendo os indices dos objetos ordenados (descendente)
9      vAuxInd = np.argsort(vetValObj)[::-1]
10
11     for i in range(numObj):
12         #percorrendo os objetos
13         for j in range(1, numMoc):
14             #verifica se o objeto cabe na mochila
15             if (vetPes[j-1] + vetPesObj[vAuxInd[i]]) <= vetCapMoc[j-1]:
16                 sol[vAuxInd[i]] = j
17                 vetPes[j-1] += vetPesObj[vAuxInd[i]]
18                 break #se alocou o objeto na mochila, sai
19
20     return sol
```

A linha 1 indica que, para implementar a heurística gulosa, optou-se por utilizar algumas funções disponibilizadas pela biblioteca **NumPy**. Na linha 3 foi realizada a declaração da função e a definição dos dados de entrada, que são: o número de mochilas

e objetos, os valores e os pesos dos objetos e a capacidade das mochilas. Na linha 4 foi declarado um vetor para armazenar a solução que será gerada de forma gulosa, na linha 6 foi declarado o vetor responsável por armazenar o peso alocado em cada mochila, isso será necessário, pois ao alocar os itens de maior valor nas mochilas, a heurística deverá verificar se a capacidade ainda comporta o item que está sendo alocado.

A linha 9 é responsável pela ordenação e precisa ser analisada com cuidado, primeiro note que foi invocada a função **argsort()**, que conforme visto na aula 7 tem a capacidade de retornar os índices relativos dos itens ordenados, mas, porque os índices? Bem, estão disponíveis os dados dos itens (valor e peso) armazenados em vetores e a referência de busca para os itens é o índice da posição no vetor, então o que a heurística precisa, não é dos itens ordenados, mas dos índices relativos da posição deles ordenados. Além disso, é necessário que estejam em ordem decrescente, pois os maiores valores devem ocupar as primeiras posições, por isso o retorno da função foi configurado com o código **[::-1]**, por fim, observe que foi passado como argumento para a função **argsort()** a lista de valores dos objetos, pois como já mencionado, neste problema, o interesse é nós itens de maior valor.

Após obter os índices e armazená-los em *vAuxInd* foi iniciado um laço que percorre os objetos, na linha 11 e para cada objeto é realizado outro laço que percorre as mochilas, na linha 13. O objetivo é verificar se o objeto cabe na mochila, na linha 15, se couber então ele é alocado, conforme a linha 16 e o peso do objeto é adicionado na mochila, linha 17, para que na próxima verificação a mochila já tenha o peso contabilizado do item que foi inserido. O comando **break** na linha 18 tem a função de interromper o laço que percorre as mochilas, caso o item já tenha sido alocado em uma mochila, isso evitará que o item seja alocado em mais de uma mochila. Por fim, a linha 20 é o retorno da solução para o método que invocou a heurística gulosa. Para testar a heurística gulosa será necessário ajustar o programa principal para, ao invés de invocar a heurística aleatória, fazer a invocação da heurística gulosa. Desta forma, faça o ajuste conforme o código disponibilizado a seguir.

```
1  import leDados as LD
2  import heuConGul as HCG
3  import calcFO as CF
4
5  numObjetos = numMochilas = 0
6  vetValoresObjetos = []
7  vetPesosObjetos = []
8  vetCapacidadeMochilas = []
9  numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas = LD.leDados()
10
11 #obtendo uma solucao construtiva gulosa
12 sol = HCG.heuConstrutivaGulosa(numObjetos, numMochilas, vetValoresObjetos,
       vetPesosObjetos, vetCapacidadeMochilas)
13
14 #calculando a FO da solucao obtida
15 FO = CF.calcFO(sol, numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas)
16
17 print "Solucao: ", sol
18 print "FO: ", FO
```

Os ajustes realizados são poucos, na linha 2, foi trocada a importação do arquivo de código-fonte da heurística aleatória pelo da heurística gulosa e na linha 12, foi ajustado

a invocação da função para a nova heurística implementada. O restante do código permaneceu inalterado. Agora é possível testar a heurística gulosa da mesma forma que a heurística aleatória, conforme a seguir:

```
>>> runfile('C:/AulaPython/prgMain.py', wdir='C:/AulaPython')
Solucao:  [2, 1, 2, 0, 1, 1, 2, 2, 0, 2, 0, 0, 2, 2, 2, 1, 1, 1, 2, 1]
FO:  10357
```

A heurística gulosa retornou o valor da *FO* em 10.357, pode-se observar também pela solução que, o item 1 não coube na mochila 1, pois foi alocado na mochila 2, já o item 2 coube na mochila 1, na sequência o item 3 também não coube na mochila 1 e foi alocado na 2, o item 4, por sua vez não coube em nenhuma das mochilas e assim os itens foram sendo alocados ou não até que todos foram analisados. Uma característica interessante que deve ser destacada é que, se você executar a heurística gulosa várias vezes, notará que a solução e o *FO* retornados são sempre iguais, isso é natural, pois para os dados que estamos utilizando nos testes, a ordenação será sempre a mesma, então é de se esperar que o resultado seja sempre o mesmo, a menos que os dados sejam alterados. Por outro lado a heurística aleatória retornará sempre soluções diferentes, apesar de possível, é pouco provável que seja sorteada uma solução igual, assim, neste caso, haverá momentos em que a heurística aleatória poderá retornar uma solução melhor ou pior que a heurística gulosa.

9.7 Heurísticas de Refinamento

As heurísticas de refinamento, também conhecidas como **busca local**, são técnicas que variam desde simples algoritmos construtivos e iterativos à métodos complexos que requerem ajustes finos e complexos, como é caso dos algoritmos evolutivos. A ideia da busca local é simples: comece com uma solução e melhore ela fazendo mudanças locais até que não haja mais progresso (GONZALEZ, 2007, p. 33). Quando utilizamos uma heurística ou meta-heurística para buscar uma solução viável, não há como determinar que a solução encontrada será um ótimo local ou global, ou ainda, não há como saber o quão próxima esta solução está de um ponto ótimo no espaço de soluções. Desta forma, como exemplificado na figura 14, a solução encontrada pela heurística, ponto azul, pode ou não estar muito próxima de um ponto de ótimo no espaço de soluções, ponto vermelho superior, assim, o objetivo da busca local é permitir que a região vizinha seja explorada, dando condições para que este ponto possa ser encontrado, caso ele esteja próximo.

Desta forma, a ideia geral do funcionamento da busca local é: a cada iteração que resulte em uma solução representada pelo ponto azul na figura 14, sejam realizadas buscas adicionais na região de vizinhança, representada pelas setas tracejadas, até um determinado ponto, representado pelos pontos vermelhos. Existem várias estratégias na realização da busca local, cada uma com padrões distintos que irão produzir buscas locais mais ou menos intensas. Determinar qual delas utilizar está associado ao domínio que temos do problema, pois uma heurística que explora um espaço amostral pequeno, não necessariamente vai obter resultados ruins, isso depende do problema que está sendo tratado (GOLDBARG; GOLDBARG; LUNA, 2016, p. 78).

Mas quem são os vizinhos? O mais comum, é que os vizinhos sejam determinados pelo critério do afastamento, ou seja, dado um parâmetro que delimita o quanto pode-se afastar da solução inicial, todas as soluções naquele "raio"de abrangência serão consideradas vizinhas. Desta forma, Goldbarg, Goldbarg e Luna (2016, p. 78) definem vizinhança, assim: "pode-se definir vizinhança $N(s,\sigma)$ de uma solução s como o conjunto

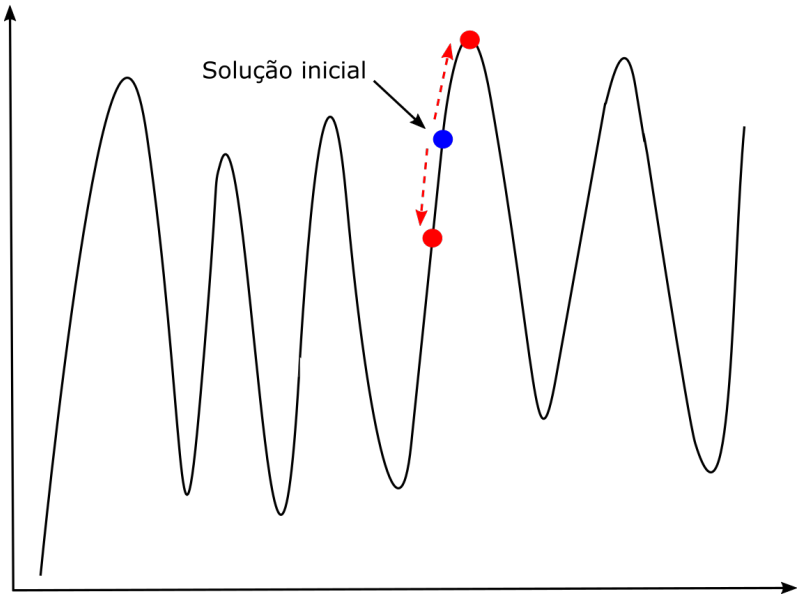

Figura 14 – Busca local

de soluções que podem ser alcançadas a partir de s por meio da operação definida por σ". Para exemplificar, no problema do caixeiro viajante, a vizinhança de uma solução pode ser definida pelo conjunto de todas as soluções do problema que diferem de s por duas arestas. Na sequência, serão implementadas algumas destas heurísticas.

9.7.1 Método da melhor melhora

O método da melhor melhora, também conhecido como *best improvement*, escolhe em cada etapa uma das posições de busca na vizinhança que resulta em uma melhoria máxima possível do valor da função objetivo. Os laços podem ser quebrados aleatoriamente, com base na ordem em que o vizinho é examinado ou usando determinados critérios (GONZALEZ, 2007, 319). O critério de parada, geralmente está associado ao fato de que o método encontra o ótimo local, ou seja, quando não é mais possível melhor a solução encontrada. A seguir foi disponibilizada uma possível implementação para a heurística da melhor melhora para o PMM, siga os passos:

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Cole o código em um editor de textos plano de sua preferência;
3. Salve o arquivo com o nome: **heuMM.py** na pasta **C:\AulaPython**.

```
1  import calcFO as CF
2
3  def heuMelhorMelhora(solucao, numObj, numMoc, valObj, vetPesObj, vetCapMoc):
4      while 1:
5          #obtem a FO da solucao atual
```

```
6         mFO = CF.calcFO(solucao, numObj, numMoc, valObj, vetPesObj, vetCapMoc)
7         #inicializa as variaveis que vao armazenar
8         #o controle de troca de objeto e mochila
9         mO = -1
10        mM = -1
11
12        for i in range(numObj):
13            mAtual = solucao[i] #guarda a mochila atual
14            for j in range(numMoc):
15                solucao[i] = j #altera a mochila e calcula a FO da nova solucao
16                FOAtual = CF.calcFO(solucao, numObj, numMoc, valObj, vetPesObj,
                      vetCapMoc)
17                if FOAtual > mFO: #se melhorar entao guarda as posicoes
18                    mO = i
19                    mM = j
20                    mFO = FOAtual
21
22            solucao[i] = mAtual #restaura a mochila atual
23
24        if mO <> -1: #se verdadeiro entao houve melhora
25            solucao[mO] = mM #guarda a melhora que ocorreu
26        else:
27            break #se nao tem como mais melhorar, termina
28
29    return solucao
```

Na linha 3 do código-fonte pode-se observar a definição da função que executará o funcionamento da heurística de melhor melhora, os parâmetros de entrada são os mesmos da função de cálculo de *FO*. Veja que a primeira linha do código da função, a linha 4, é um laço cuja condição é **1**, ou seja, é sempre verdadeiro, pois o objetivo é que o laço execute ao menos uma vez e o controle de interrupção foi incluído no final de suas instruções, no caso, entre as linhas 24 e 27. A primeira instrução do laço **while** é o cálculo da *FO* da solução inicial, pois esta heurística requer uma solução inicial como ponto de partida.

Entre as linhas 12 e 22 é executado um laço que percorre cada objeto e para cada objeto é feita a troca de mochilas em que o objeto foi alocado, ao fazer uma troca é realizado o cálculo da *FO* na linha 16, para verificar se a *FO* melhorou em relação à anterior, caso tenha melhorado, guarda as posições do objeto e da mochila, cujo valor resultou em melhor. Após fazer todas as trocas de mochila para o item atual, a mochila em que o item estava alocado inicialmente é restaurada na linha 22, pois o objetivo é fazer uma troca apenas, a melhor, em cada iteração do laço **while**. A troca ocorre na linha 25, caso a busca tenha encontrado alguma solução melhor que a atual, caso contrário o processo é interrompido, pois não é mais possível melhorar. Para testar a heurística de melhor melhora, é necessário ajustar o programa principal, conforme o código a seguir:

```
1 import leDados as LD
2 import heuConAle as HCA
3 import calcFO as CF
4 import heuMM as MM
5
6 numObjetos = numMochilas = 0
7 vetValoresObjetos = []
8 vetPesosObjetos = []
```

```
9  vetCapacidadeMochilas = []
10 numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas = LD.leDados()
11
12 #obtendo uma solucao construtiva gulosa
13 sol = HCA.heuConstrutivaAleatoria(numObjetos, numMochilas)
14
15 #calculando a FO da solucao obtida
16 FO = CF.calcFO(sol, numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas)
17
18 print "Solucao: ", sol
19 print "FO: ", FO
20
21 #realizando a melhor melhora
22 sol = MM.heuMelhorMelhora(sol, numObjetos, numMochilas, vetValoresObjetos,
       vetPesosObjetos, vetCapacidadeMochilas)
23 #calculando a FO da solucao obtida
24 FO = CF.calcFO(sol, numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas)
25
26 print "Solucao: ", sol
27 print "FO: ", FO
```

O primeiro ajuste é a importação do arquivo de código-fonte da heurística de melhor melhora na linha 4. Observe que foi utilizada a heurística aleatória para gerar a solução inicial na linha 13. Após isso foi impressa a solução e a *FO* da solução gerada aleatoriamente, para comparação posterior. Na linha 22 foi invocada a função da heurística de melhor melhora e novamente foram impressos, a solução e a *FO*, nas linhas seguintes. A seguir a execução do programa principal.

```
1 >>> runfile('C:/AulaPython/prgMain.py', wdir='C:/AulaPython')
2 Solucao:  [1, 1, 3, 0, 3, 3, 0, 1, 3, 2, 1, 1, 1, 3, 3, 0, 2, 1, 0, 1]
3 FO:  -153280
4 Solucao:  [1, 1, 3, 0, 3, 3, 0, 2, 3, 2, 2, 1, 1, 3, 3, 2, 2, 1, 0, 1]
5 FO:  10554
```

No teste realizado para o livro a solução gerada aleatoriamente é inviável, isso pode ser observado na linha 3, pois a mesma foi penalizada. Contudo, ao executar a heurística de melhor melhora, foi possível obter uma solução viável com uma *FO* muito superior à anterior, como pode ser visto na linha 5. Analisando as soluções encontradas, na linha 2 e 4, percebe-se que as duas são quase iguais, apenas alguns itens foram trocados de mochila, o que mostra que busca na vizinhança foi eficiente em encontrar, não só uma solução viável, mas com um possível ótimo local [2].

9.7.2 Método da primeira melhora

O método da primeira melhora, também denominado como *first improvement* examina o vizinho em alguma ordem pré-definida e executa a primeira etapa de busca pela melhor solução, uma vez encontrada uma solução com "melhor"*FO*, a busca local

[2] Como já mencionado, não é possível determinar se a solução encontrada por uma heurística é ou não um ótimo local ou global, exceto nos casos em que foi possível obter o ótimo global pelo método exato, pois assim, seria possível comparar com os resultados da heurística.

é interrompida. Claramente, o ótimo local encontrado por este método depende da ordem em que o vizinho é sondado. Em vez de usar ordens predefinidas e fixas, pode ser benéfico examinar a vizinhança em ordem aleatória, e as buscas repetidas de algoritmos de primeira melhora em modo aleatório, geralmente são capazes de identificar muitos ótimos locais diferentes, mesmo quando iniciadas a partir da mesma solução inicial (GONZALEZ, 2007, 319). O método da primeira melhora é uma alternativa ao método da melhor melhora que requer o exame de toda a vizinhança, pois, isto demanda muito processamento, o que pode vir a penalizar a performance da heurística. No método da primeira melhora, essa sequência de busca é interrompida quando um melhor vizinho é encontrado, melhorando muito a performance deste método se comparado à melhor melhora.

```
1  import calcFO as CF
2
3  def heuPrimeiraMelhora(solucao, numObj, numMoc, valObj, vetPesObj, vetCapMoc):
4      while 1:
5          #obtem a FO da solucao atual
6          mFO = CF.calcFO(solucao, numObj, numMoc, valObj, vetPesObj, vetCapMoc)
7          #inicializa a variavel de controle da interrupcao da busca
8          melhorou = False
9
10         for i in range(numObj):
11             mAtual = solucao[i] #guarda a mochila atual
12             for j in range(numMoc):
13                 solucao[i] = j #altera a mochila e calcula a FO com a nova
                       solucao
14                 FOAtual = CF.calcFO(solucao, numObj, numMoc, valObj, vetPesObj,
                       vetCapMoc)
15                 if FOAtual > mFO: #se melhorar entao guarda as informacoes
16                     melhorou = True #registra q melhorou
17                     mFO = FOAtual #guarda a melhor FO
18                     break #sai do laco de mochilas
19
20             if melhorou:
21                 break #sai do laco de objetos
22             else:
23                 solucao[i] = mAtual #restaura a mochila atual
24
25         if not melhorou: #se nao melhorou interrompe
26             break
27
28     return solucao
```

O código-fonte para o método da primeira melhora é muito parecido com o código da melhor melhora, pois a única diferença, é que, na melhor melhora o método avaliará a troca com todas as mochilas e no método da primeira melhora, se houver uma troca de mochila em que o valor da *FO* seja melhor, então ele deve parar naquele ponto e iniciar a avaliação do próximo item. Na linha 8 do código foi declarada uma variável, *melhorou*, para fazer este controle, observe que ela é inicializada com **False** a cada iteração do laço **while** e caso seja feita uma troca que resulte em melhor *FO*, então a variável *melhorou* recebe **True** na linha 16, para que na linha 20 o laço seja interrompido. Para testarmos o método, é preciso ajustar o programa principal novamente, conforme o código a seguir.

```
1  import leDados as LD
2  import heuConAle as HCA
3  import calcFO as CF
4  import heuPM as PM
5
6  numObjetos = numMochilas = 0
7  vetValoresObjetos = []
8  vetPesosObjetos = []
9  vetCapacidadeMochilas = []
10 numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas = LD.leDados()
11
12 #obtendo uma solucao construtiva gulosa
13 sol = HCA.heuConstrutivaAleatoria(numObjetos, numMochilas)
14
15 #calculando a FO da solucao obtida
16 FO = CF.calcFO(sol, numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas)
17
18 print "Solucao: ", sol
19 print "FO: ", FO
20
21 #realizando a melhor melhora
22 sol = PM.heuPrimeiraMelhora(sol, numObjetos, numMochilas, vetValoresObjetos,
       vetPesosObjetos, vetCapacidadeMochilas)
23 #calculando a FO da solucao obtida
24 FO = CF.calcFO(sol, numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas)
25
26 print "Solucao: ", sol
27 print "FO: ", FO
```

A linha 4 foi ajustada para importar o arquivo de código-fonte do método da primeira melhora e a linha 22 foi ajustada para invocar a função da primeira melhora. O restante do código-fonte é idêntico. Veja a seguir o resultado do teste feito para o livro. Vale lembrar que estamos utilizando a heurística aleatória para gerar a solução inicial, de forma que, se você executar em seu computador, provavelmente vai obter valores diferentes.

```
1  >>> runfile('C:/AulaPython/prgMain.py', wdir='C:/AulaPython')
2  Solucao:  [1, 0, 3, 2, 0, 3, 0, 3, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 2, 0]
3  FO:  -164056
4  Solucao:  [2, 2, 3, 2, 2, 3, 2, 3, 2, 2, 0, 2, 1, 1, 1, 1, 1, 1, 2, 1]
5  FO:  11232
```

No teste realizado, novamente a heurística construtiva aleatória produziu uma solução inviável na linha 2, pois a *FO* foi penalizada conforme a linha 3. Contudo, a heurística da primeira melhora foi capaz de, a partir da solução inicial, obter uma melhor solução viável, conforme as linhas 4 e 5.

9.7.3 Método de Descida/Subida Randômica

O método da descida/subida randômica, também denominado como *Random Descent* ou *Uphill Method*, de forma similar aos métodos já vistos, requer uma solução inicial,

e a cada passo seleciona um vizinho aleatoriamente, após isso, aceita apenas se ele for melhor que a solução atual, caso contrário a solução atual permanece inalterada e um novo vizinho é gerado. Assim, este método é uma alternativa ao método da melhor melhora, pois enquanto a melhor melhora requer a análise exaustiva de todos os vizinhos, o método da descida/subida randômica não adota este critério, analisando um vizinho qualquer, desta forma, é necessário que seja estabelecido um critério de parada, que geralmente é um número fixo de iterações sem melhorar o valor obtido até então.

9.7.4 Método Não Ascendente/Descendente Randômico

O método Não Ascendente Randômico (RNA) ou método Não Descendente Randômico (RND) é uma variação do método de descida/subida randômica, com apenas uma diferença, este método aceita o vizinho gerado aleatoriamente se ele for melhor ou igual à solução corrente. A estratégia de parada também é igual ao método da descida/subida randômica. Como este método aceita o vizinho com *FO* igual, então ele permite a navegação no espaço de busca com movimentos laterais. Possibilitando assim percorrer caminhos de descida/subida que passam por regiões planas. O método RNA/RND é, portanto, um procedimento que explora o espaço de soluções combinando movimentos de descida/subida com movimentos laterais.

9.7.5 Método da Descida em Vizinhança Variável

De acordo com Hertz e Mittaz (2001, p. 428), o método da descida em vizinhança variável, é uma variação do método *Variable Neighborhood Search* (VNS) que, por sua vez, foi proposto por Mladenović e Hansen (1997). Esta variação, também denominada como *Variable Neighborhood Descent (VND)*, explora os vizinhos distantes da atual solução, e passa para uma nova solução, se e somente se, uma melhoria ocorrer, de modo que, o método de busca local é aplicado repetidamente para obter soluções vizinhas com ótimo local (HERTZ; MITTAZ, 2001, p. 428). O pseudo-código para o VND é apresentado a seguir:

```
1  procedimento VND(S, K_max)
2  repita
3      k ← 1
4      enquanto k ≤ K_max faca
5          encontrar a melhor vizinhanca s' de S,(s' ∈ N'_k(S))
6          se f(s') < f(S) entao
7              S ← s'
8              k ← 1
9          senao
10             k ← k+1
11         fimse
12     fimenquanto
13 ate nenhum melhoramento ser obtido
14 retorna S
```

Fonte: Adaptado de (LOPES; RODRIGUES; STEINER, 2013, p. 149)

No código base apresentado, o conjunto de vizinhanças é definido por $N'_k, k = 1,2,...,k'_{max}$ e S é a solução inicial. Observe que, caso a solução vizinha seja melhor que a solução atual, conforme a linha 6 do código, então a variável k recebe novamente o valor **1**, isso vai garantir que a vizinha continue a ser explorada, caso soluções melhores

estejam sendo encontradas. Dependendo do problema, a busca pelo ótimo local pode ser cara computacionalmente, assim, ao aplicar este método, é muito comum considerar a exploração apenas em um certo percentual da vizinhança, isto é, procurar o melhor vizinho somente em um dado percentual que é determinado como um parâmetro do método. No pseudo-código apresentado, esse parâmetro é representado por K_{max}.

9.8 Heurísticas de Intensificação

As heurísticas de intensificação tem o objetivo de concentrar a busca em regiões que são consideradas promissoras, pois para muitos problemas, ótimos locais são relativamente próximos de ótimos globais. Assim, estes métodos são utilizados em conjunto com alguma heurística ou meta-heurística e, naturalmente, serão aplicados em algum momento da execução da heurística, em que for identificada uma área promissora.

9.8.1 Reconexão por Caminhos

A reconexão por caminhos, também denominada como *Path Relinking* é uma estratégia de criação de trajetórias de movimentos entre soluções "elite", ou seja, soluções com ótimos locais. A estratégia tem o objetivo de integrar a intensificação e diversificação. A reconexão por caminhos geralmente opera começando a partir de uma solução inicial, selecionada de um subconjunto de soluções de alta qualidade e gerando um caminho no espaço de vizinhança que leva às outras soluções no subconjunto, que são chamadas de soluções orientadoras. A abordagem é chamada de reconexão por caminhos, em virtude de gerar um novo caminho entre soluções previamente vinculadas por uma série de movimentos executados durante uma pesquisa ou gerando um caminho entre soluções anteriormente vinculadas a outras soluções, mas não umas às outras (GONZALEZ, 2007, p. 384).

Esta estratégia, geralmente é computacionalmente cara, assim, deve ser utilizada apenas em determinados períodos ao longo da execução da heurística, por exemplo, após um certo número de iterações. A seguir foi disponibilizado o pseudo-código do **Path Relinking**.

```
1  Ler entrada de dados
2  Calcular Δ(S_i, S_f)
3  fx ← min{f(S_i), f(S_f)}
4  Sx ← argmin{f(S_i), f(S_f)}
5  S ← S_i
6  Enquanto Δ(S_i, S_f) ≠ ∅ Faca
7      mx ← argmin{f(S_i ⊕ m), m ∈ Δ(S_i, S_f)}
8      Δ(S ⊕ mx, S_f) ← Δ(S_i, S_f) \ {mx}
9      S ← S ⊕ mx
10     Se f(S) < fx entao
11         fx ← f(S)
12         Sx ← S
13     fimse
14 fimenquanto
15 Retorne com a solucao fx
```

Fonte: adaptado de Goldbarg, Goldbarg e Luna (2016, p. 103)

A solução S_i é a solução inicial do método, e S_f é a solução que pretende-se atingir ao final do procedimento. A função $\Delta(S_i, S_f)$ na linha 2, é responsável por obter a diferença simétrica que é o conjunto de movimentos necessários para, a partir da solução inicial,

alcançar a solução final. A letra m representa um movimento pertencente ao conjunto de movimentos calculado pela diferença simétrica, mx refere-se ao melhor movimento obtido. A função $f(S)$ é responsável por calcular o valor da FO (GOLDBARG; GOLDBARG; LUNA, 2016, p. 103).

9.8.2 Principio da Otimalidade Próxima (POP)

O principio da otimalidade próxima (POP), também denominado como *Proximate Optimality Principle*, é outra estratégia que se baseia na ideia de que "boas soluções"em um determinado nível, provavelmente sejam encontradas "perto de"boas soluções em um nível adjacente. A seguir um exemplo da aplicação de **POP** para o PMM com 5 itens.

Exemplo de aplicação do POP

1. Cria-se uma solução inicial com os 3 primeiros itens;
2. Aplica-se uma heurística de refinamento nesta solução;
3. Insere-se o quarto item na solução resultante do refinamento;
4. Aplica-se uma heurística de refinamento nesta solução;
5. Insere-se o quinto item na solução resultante do refinamento;
6. Aplica-se uma heurística de refinamento nesta solução.

Devido a necessidade de eficiência, uma implementação prática de **POP** para alguma meta-heurística pode ser: aplicar uma busca local durante alguns pontos na fase de construção e não durante cada iteração de construção. Por exemplo, a busca pode ser aplicada após 40% e 80% em que movimentos de construção foram realizados, bem como no final da fase de construção (GLOVER; KOCHENBERGER, 2003, p. 231).

9.9 Resumo da Aula

As primeiras implementações de heurísticas com a linguagem Python, aplicadas ao problema da mochila múltipla foram desenvolvidas nesta aula. Assim, foi dado um passo importante para as próximas aulas, pois, foi desenvolvida a base do programa que será utilizada com as meta-heurísticas, pois operações como: a leitura dos dados, as heurísticas construtivas, as heurísticas de busca local, o cálculo da função objetivo, e o programa principal serão reutilizados nas próximas aulas.

Foram implementados os códigos da heurística construtiva aleatória, da heurística construtiva gulosa, a heurística de melhor melhora e a heurística da primeira melhora. Para alguns métodos que foram apresentados, não foi disponibilizada a implementação para oportunizar ao leitor esta experiência, como é o caso do método de descida/subida randômica, o método não ascendente/descendente randômico e o método da descida em vizinhança variável. Por fim, foram apresentados também alguns métodos de intensificação que tem o objetivo de intensificar a busca local em regiões promissoras.

9.10 Exercícios da Aula

1. Faça um programa em Python para otimizar a alocação de itens na mochila com os dados disponibilizados no início da aula 9. Utilize a heurística construtiva gulosa para gerar a solução inicial e a heurística da descida/subida randômica como estratégia de busca local.

2. Ajuste o programa construído para o exercício anterior, mudando a estratégia de busca local para a heurística do método não ascendente/descendente randômico.

3. Ajuste o programa construído no exercício anterior, mudando a busca local para o método da descida em vizinhança variável.

AULA 10

Meta-heurística GRASP

Metas da Aula

1. Compreender o modo de funcionamento da meta-heurística GRASP.
2. Implementar a meta-heurística GRASP em linguagem Python para o problema da mochila múltipla (PMM).
3. Realizar testes da meta-heurística GRASP em conjunto com heurísticas de refinamento.

Ao término desta aula, você será capaz de:

1. Implementar a meta-heurística GRASP em linguagem Python.
2. Entender como aplicar a meta-heurística GRASP em um problema a partir de um pseudo-código.

10.1 Meta-heurística GRASP

O nome **GRASP** refere-se à *Greedy Randomized Adaptive Search Procedure* que em português equivale à Procedimento de busca adaptativa gulosa e randômica. É uma meta-heurística que foi inicialmente descrita por Feo e Resende (1989), trata-se de um processo iterativo e *multi-start*, pois tem vários inícios ao longo de seu ciclo (GLOVER; KOCHENBERGER, 2003, p. 219). Além disso, combina heurística construtiva e de refinamento, pois inclui busca local. A fase de construção é realizada de forma semigulosa e adaptativa, cujo objetivo é construir uma solução viável. Já a fase seguinte, envolve fazer uma busca local, tendo como solução inicial a produzida pela etapa anterior. As heurísticas gulosas, no geral, não são eficientes em amostrar adequadamente o espaço de busca, assim, a combinação das duas fase do GRASP ataca os problemas de qualidade das configurações e de diversificação das soluções geradas (GOLDBARG; GOLDBARG; LUNA, 2016, p. 98).

O GRASP é uma meta-heurística formada basicamente por um procedimento construtivo, cuja estratégia é gulosa aleatorizada, e uma busca local não especificada, ou seja, a estratégia de busca local pode variar conforme a necessidade e o problema. A fase de construção também permite a adoção de variadas estratégias, mas a mais conhecida é a semigulosa proposta por Hart e Shogan (1987). Como esta fase envolve gulosidade, certos problemas irão permitir a ordenação dos elementos, de forma, a possibilitar a obtenção do item de "maior valor"associado, contudo, no GRASP, não necessariamente isso irá ocorrer, pois esta etapa é flexibilizada por meio de uma lista restrita de candidatos (*LRC*), assim, apenas candidatos pertencentes à esta lista irão compor a solução na fase inicial. A *LRC* é um parâmetro do GRASP que irá determinar quantos candidatos vão fazer parte da lista (GOLDBARG; GOLDBARG; LUNA, 2016, p. 99).

10.2 Algoritmo Base do GRASP

```
1  procedimento GRASP(max_iteracoes, LRC)
2  mSolucao ← ∞
3  para i de 1 ate max_iteracoes faca
4      solucao ← construtiva_aleatoria_gulosa(LRC)
5      solucao ← busca_local(solucao)
6      se (f(solucao) < f(mSolucao))
7          mSolucao ← solucao
8      fimse
9  fimpara
10 retorna mSolucao
```

Fonte: Adaptado de (GLOVER; KOCHENBERGER, 2003, p. 220)

Na linha 1 do pseudo-código foi declarado o procedimento GRASP, observe que são dois argumentos de entrada, *max_iteracoes* que refere-se ao número máximo de iterações que devem ser executadas, que também pode ser estabelecido por tempo de duração em segundos, minutos, etc. Na linha 2 foi declarada *mSolucao*, cujo objetivo é armazenar a melhor solução encontrada pelo GRASP. *mSolucao* recebeu o valor infinito, pois assim, na primeira comparação, na linha 6, entre a solução gerada pelo GRASP e pela melhor solução, até o momento, a solução atual será sempre melhor, garantindo assim que a primeira solução seja registrada. Esta configuração é para um problema de minimização, caso o problema seja de maximização, então *mSolucao* deve receber $-\infty$ e na linha 6 a comparação deve ser $f(solucao) > f(mSolucao)$.

Na linha 4, em cada iteração, será gerada uma solução por meio de um método aleatório guloso, com base na *LRC*, na linha 5 é aplicada uma busca local na solução construída no passo anterior. Após isso, na linha 6 é feita uma comparação para verificar se a solução gerada neste iteração é melhor que a solução gerada anteriormente, caso seja, então ela é aceita, conforme a linha 7. Ao final do procedimento, *mSolucao* terá armazenada a melhor solução encontrada pelo GRASP, assim, ela é retornada para o método que a invocou conforme a linha 10.

10.3 Implementando a meta-heurística GRASP para o PMM

A tarefa de implementar o GRASP para o PMM, já definido na aula 8, ficou muito fácil, pois boa parte do código já foi implementado na aula 9, e como a busca local do GRASP pode ser determinada conforme a necessidade e o problema, então é possível utilizar uma das buscas locais já implementadas, como o **método da melhor melhora** ou da **primeira melhora**. A primeira função que deve ser implementada, é a responsável pela heurística aleatória gulosa, pois como pode ser observado na linha 4 do pseudo-código, esta heurística é uma função acessória que será necessária na implementação do GRASP. Desta forma, siga os passos a seguir para implementar a função responsável pela heurística construtiva aleatória gulosa:

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Cole o código em um editor de textos plano de sua preferência;
3. Salve o arquivo com o nome: **heuConAleGul.py** na pasta **C:\AulaPython**.

```
1  import numpy as np
2  import random as rand
3
4  #gerando a semente
5  rand.seed()
6
7  def heuConAleGul(lrc, numObj, numMoc, vetValObj, vetPesObj, vetCapMoc):
8      #montando a LRC
9      vetAux = [0] * numObj
10     vetVal = [0] * lrc
11     vetLrc = [0] * lrc
12     for i in range(lrc):
13         obj = rand.randint(0, numObj-1)
14         while vetAux[obj] == 1: #impede o sorteio de um numero repetido
15             obj = rand.randint(0, numObj-1)
16
17         vetLrc[i] = obj
18         vetVal[i] = vetValObj[obj]
19         vetAux[obj] = 1
20
21     #inserir os objetos da LRC na mochila de forma gulosa
22     #obtendo os indices dos objetos ordenados (descendente)
23     vAuxIndLRC = np.argsort(vetVal)[::-1]
```

```
24      sol = [0] * numObj
25      vetPes = [0] * numMoc
26      for i in range(lrc):
27          for j in range(numMoc):
28              if (vetPes[j] + vetPesObj[vetLrc[vAuxIndLRC[i]]]) <= vetCapMoc[j]:
29                  sol[vetLrc[vAuxIndLRC[i]]] = j
30                  vetPes[j] = vetPes[j] + vetPesObj[vetLrc[vAuxIndLRC[i]]]
31                  break
32
33      #inserir os demais objetos na mochila de forma gulosa
34      vAuxIndObj = np.argsort(vetValObj)[::-1]
35      for i in range(numObj):
36          if vetAux[vAuxIndObj[i]] == 0: #se verdadeiro, entao nao foi alocado
            ainda
37              for j in range(numMoc):
38                  if (vetPes[j] + vetPesObj[vAuxIndObj[i]]) <= vetCapMoc[j]:
39                      sol[vAuxIndObj[i]] = j
40                      vetPes[j] = vetPes[j] + vetPesObj[vAuxIndObj[i]]
41                      break
42
43      return sol
```

A heurística construtiva aleatória e gulosa teve que ser implementada, pois ela deve construir a solução com base na *LRC*, que é uma particularidade do GRASP, além disso, cada problema demandará um tratamento diferente. Para o PMM, esta implementação aqui apresentada, é uma possível solução, dentre várias. Se observar atentamente o código, perceberá que essa heurística é uma adaptação da heurística construtiva gulosa que foi implementada na aula 9, a diferença, é que neste caso, precisamos considerar a lista restrita de candidatos, *LRC*. Veja que entre as linhas 8 e 19, foi construida a lista *LRC* de forma aleatória. Na linha 12 foi realizado um laço com o número de iterações do tamanho da *LRC* e para cada iteração é sorteado um item para compor a lista. O laço **while** na linha 14 é para garantir que em cada iteração seja sorteado um objeto que não tenha sido sorteado ainda. Após sortear um objeto, na linha 17, este é armazenado na lista *LRC*, na linha 18 é registrado o valor do objeto que foi sorteado e na linha 19, o objeto é marcado, como já tendo sido sorteado.

Após construir a lista *LRC*, entre as linhas 21 e 31, a solução é registrada em *sol*, de forma gulosa apenas com os itens que foram incluídos na lista restrita de candidatos. Observe que antes de incluir, é realizada uma validação da capacidade atual da mochila, na linha 28, se o objeto couber então ele é armazenado na mochila, na linha 29, depois o peso do objeto é adicionado ao peso da mochila, na linha 30, e o laço é interrompido para evitar que o objeto seja armazenado em outra mochila, na linha 31.

Após armazena os itens da *LRC* na solução, o procedimento é novamente realizado, entre as linhas 33 e 41, mas agora com o objetivo de alocar o restante dos objetos. Observe que o processo é praticamente igual ao anterior, a diferença é que em cada objeto visitado no laço da linha 35, é realizada uma verificação para validar se o objeto não foi armazenado em alguma mochila, na linha 36, caso não tenha sido, então é um laço percorre as mochilas, na linha 37 para verificar se o objeto cabe em alguma e alocá-lo, se for o caso. Ao terminar este último laço pelos objetos, a solução estará completa e pronta para ser devolvida ao método que invocou a função, conforme a linha 43. O próximo passo é a implementação do método GRASP, faça:

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Cole o código em um editor de textos plano de sua preferência;
3. Salve o arquivo com o nome: **grasp.py** na pasta **C:\AulaPython**.

```
1  import time
2  import calcFO as CF
3  import heuConAleGul as HCG
4  import heuPM as PM
5
6  def grasp(lrc, tempo, numObj, numMoc, vetValObj, vetPesObj, vetCapMoc):
7      ini = time.time()
8      achouT = time.time()
9      melhorFO = float("-inf") #obtendo valor infinito negativo
10     lrc = (lrc * numObj) / 100 #calculando o % da LRC com base nos objetos
11     mSol = [0] * numObj
12
13     while 1:
14         sol = HCG.heuConAleGul(lrc, numObj, numMoc, vetValObj, vetPesObj,
               vetCapMoc)
15
16         #busca local
17         sol = PM.heuPrimeiraMelhora(sol, numObj, numMoc, vetValObj, vetPesObj,
               vetCapMoc)
18         FO = CF.calcFO(sol, numObj, numMoc, vetValObj, vetPesObj, vetCapMoc)
19
20         if FO > melhorFO:
21             mSol = sol
22             melhorFO = FO
23             achouT = time.time()
24
25         fim = time.time()
26         #verifica se deve continuar executando
27         if fim <= (ini + tempo):
28             continue
29         else:
30             break
31
32     return mSol, (achouT - ini)
```

O código-fonte do GRASP em Python tem as mesmas características do código base já visto, o que muda, são as questões relacionadas à arquitetura da linguagem e os detalhes relativos ao PMM, como por exemplo, os parâmetros que são específicos do problema, número de objetos, número de mochilas, valores dos objetos, etc. O GRASP, aqui implementado, adota o tempo como estratégia para manter o laço ativo, por isso, foi importada a biblioteca com funções de tempo, na linha 1, foi também inicializada uma variável com a hora atual na linha 7 e foi verificado se o laço continua ou não, por meio do tempo, entre as linhas 25 e 30, observe que a variável *tempo* utilizada na linha 27 é um argumento da função GRASP conforme a linha 6. Na linha 9 a variável *melhorFO* recebe o valor infinito negativo, pois o PMM é de maximização. Na linha 10 a variável *lrc* é ajustada com base no percentual dos objetos, adotou-se esta estratégia,

pois o nosso código, até o presente momento tem sido implementado de forma a permitir configurações diferentes para a quantidade de objetos e mochilas, ou seja, essas informações não são fixas, então é importante converter o tamanho da *LRC* em termos percentuais.

Na linha 13 inicia o laço **while** com a condição que será sempre verdadeira, pois como já mencionado, a interrupção do laço será controlada entre as linhas 25 e 30 com base no tempo. Na linha 14 foi invocada a função responsável por gerar a solução de forma construtiva aleatória e gulosa, após obter a solução é realizada a busca local tendo esta como solução inicial, na linha 18 é invocada a função que calcula a *FO* para a solução gerada e na linha 20 é realizada a verificação se a *FO* obtida para a solução atual é melhor que a *FO* já registrada no passo anterior, que no caso da primeira execução, será o valor infinito negativo, e nas demais execuções, será realmente o maior valor observado até o momento. Se a condição da linha 20 retornar verdadeiro, então a solução, até então a melhor, será registrada na variável *mSol*, a *FO* da solução atual será registrada em *melhorFO* e o tempo será registrado em *achouT*. O objetivo de registrar o momento em que a melhor solução foi encontrada, é poder, por exemplo, avaliar se o tempo de execução do GRASP está sendo suficiente ou não, para encontrar o ótimo local. A seguir, os passos para testar o GRASP com os dados do PMM.

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Abra o arquivo **prgMain.py** presente na pasta **C:\AulaPython**;
3. Cole o código substituindo o código atual do arquivo;
4. Salve o arquivo.

```
1  import leDados as LD
2  import calcFO as CF
3  import grasp as GR
4
5  numObjetos = numMochilas = 0
6  LRC = 50 #neste exemplo adotou-se uma LRC de 50% dos objetos
7  tempoExec = 10 #neste exemplo adotou-se 10 segundos de tempo de execucao
8  vetValoresObjetos = []
9  vetPesosObjetos = []
10 vetCapacidadeMochilas = []
11 numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas = LD.leDados()
12
13 #obtendo uma solucao pela meta-heuristica GRASP
14 sol, tempo = GR.grasp(LRC, tempoExec, numObjetos, numMochilas, vetValoresObjetos,
       vetPesosObjetos, vetCapacidadeMochilas)
15
16 #calculando a FO da solucao obtida
17 FO = CF.calcFO(sol, numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas)
18
19 print "Solucao: ", sol
20 print "FO: ", FO
21 print "Achou em: %.2f " % tempo
```

Na linha 3 foi importado arquivo do código-fonte do GRASP, na linha 6 foi definida a *LRC* com 50, ou seja, pretende-se que a lista restrita de candidatos comporte 50% dos objetos. Na linha 7 foi definido o tempo como sendo 10 segundos. O GRASP foi invocado na linha 14, veja que além dos argumentos do PMM, o método recebeu também o tamanho da *LRC* e o tempo de execução, além disso, observe também que o método retorna dois valores, a solução, armazenada em *sol* e o tempo para encontrar a solução, armazenado na variável *tempo*. Na linha 17 foi calculado o valor da *FO* para a solução e por fim, os valores foram impressos entre as linhas 19 e 21. Agora é possível testar a meta-heurística GRASP, para executar abra o interpretador do Python e execute o programa principal, conforme o trecho a seguir.

```
1 >>> runfile('C:/AulaPython/prgMain.py', wdir='C:/AulaPython')
2 Solucao:  [2, 2, 2, 0, 2, 1, 2, 2, 0, 2, 0, 0, 1, 1, 1, 1, 2, 2, 1, 1]
3 FO:  10357
4 Achou em: 6.35
```

Como o GRASP foi configurado para executar por 10 segundos, então, as linhas 2, 3 e 4 vão demorar em torno de 10 segundos para aparecer. É possível que, embora tenhamos configurado para executar por 10 segundos, o tempo de execução real não seja exatamente 10 segundos, pois a saída do laço do GRASP só é verificada após executar todos os comandos, inclusive a heurística construtiva e a busca local que são operações que demandam maior processamento, o que pode levar a execução a exceder o tempo configurado.

Na linha 2 foi impressa a melhor solução encontrada pelo GRASP, na linha 3 o valor da *FO* desta solução e na linha 4 o tempo que levou para encontrar essa solução que foi de 6 segundos aproximadamente, ou seja, as soluções encontradas após esse tempo eram todas com o valor de *FO* inferior a esta. É importante destacar que, o fato de o método ter obtido a melhor solução em 6 segundos, não quer dizer que o tempo de execução possa ser reduzido para 6 segundos, pois esse tempo provavelmente vai variar nas próximas execuções, assim, para determinar o tempo, é importante executar o GRASP várias vezes, talvez dezenas, para que seja observado o desvio-padrão do tempo, e assim, inferir um valor com maior segurança.

10.4 Calibração dos parâmetros

O GRASP possui apenas 2 parâmetros, o argumento *LRC* (lista restrita de candidatos) e o número de iterações. Como já mencionado o parâmetro do número de iterações, *max_iteracoes*, é flexível e de fácil definição, pois pode ser configurado por número de iterações (ex.: 1.000, 10.000, 100.000,...) ou por tempo (ex.: 10s, 1m, 20m,...). Uma boa estratégia é programar o algoritmo para registrar o tempo que está levando para atingir o melhor "ótimo local", pois assim, será mais fácil determinar o tempo de execução ou número de iterações, pois imagine o seguinte cenário: em um certo conjunto de testes o programador executou o GRASP por 30 minutos, contudo, a solução com "ótimo local"foi encontrada em 90% dos casos com menos de 1 minuto, e em 95% dos casos foi encontrada com menos de 5 minutos, ou seja, será que é realmente necessário executar por 30 minutos? Em se tratando de métodos heurísticos, é muito comum este tipo de efeito, pois dependendo do problema, da heurística e da estratégia adotada, pode acontecer da busca ser conduzida para regiões, em que, as soluções encontradas estarão limitadas a determinados ótimos locais.

Em relação ao parâmetro *LRC*, este define o número de elementos que irão formar a solução na fase de construção, assim, se este número for muito pequeno é possível que

a solução construida não terá elementos suficientes, de forma a garantir a representatividade dos elementos amostrados para compor a *LRC*. Por outro lado, se este número é muito grande, é possível que o GRASP tenha um desempenho similar ao que seria a combinação de uma heurística construtiva aleatória com uma busca local. Desta forma, o tamanho ideal da *LRC* é algo que não se pode determinar de forma exata, e no geral, vai variar de um problema para outro, assim, a melhor forma de determinar a *LRC* é empiricamente, ou seja, com testes e análise de resultado (GLOVER; KOCHENBERGER, 2003, p. 227).

Há várias formas de compor a *LRC*, a que implementamos é denominada como: **critério de restrição por cardinalidade**, em que seleciona-se os p elementos com o menor custo. Há ainda a **restrição por valor**, em que os candidatos da lista estão contidos em um determinador intervalo de valores. O tamanho do intervalo é calculado com base em um percentual de afastamento entre um valor mínimo e um valor máximo e por meio de um parâmetro α, normalmente $\alpha \in [0,1]$ (GOLDBARG; GOLDBARG; LUNA, 2016, p. 99). A expressão a seguir mostra uma forma de implementar o controle do tamanho e dos componentes da *LRC* com o parâmetro α.

$$LRC \leftarrow \{e \in M | c(e) <= c^{min} + \alpha(c^{max} - c^{min})\} \tag{10.1}$$

Fonte: (GOLDBARG; GOLDBARG; LUNA, 2016, p. 100)

Outra forma de compor a *LRC* é denominado como **método da roleta** em que a decisão é tornada aleatória, mas considerando um sorteio de viés guloso, pois a decisão é tomada sobre uma roleta cuja área do sorteio é proporcional à qualidade das variáveis, assim, algumas variáveis terão mais chances de serem escolhidas que outras, quando a "roleta girar" (GOLDBARG; GOLDBARG; LUNA, 2016, p. 100).

10.5 Vantagens do GRASP

O GRASP pode ser facilmente adaptado e aplicado em problemas de otimização, inclusive problemas NP-difícil. Goldbarg, Goldbarg e Luna (2016, p. 102) cita que o GRASP é um método de fácil paralelização, pois, tanto o método construtivo, quanto a busca local podem ser atribuídos a processadores distintos, compartilhando, por exemplo informações das regiões do espaço de busca já visitado. Cita ainda que o algoritmo compromete pouco a memória, pois não requer o armazenamento de muitas informações das iterações realizadas. E por fim, Goldbarg, Goldbarg e Luna (2016, p. 102) citam a facilidade de modularização do GRASP, pois suas partes são de fácil separação o que favorece, por exemplo, o reuso de código e os testes, por exemplo, com variados métodos de busca local.

10.6 Resumo da Aula

Nesta aula foi discutida e implementada a primeira meta-heurística, parte do escopo deste livro, no caso, o GRASP, que é a abreviação de *Greedy Randomized Adaptive Search Procedure*. Como discutido ao longo da aula, o GRASP é uma meta-heurística simples em suas características e também na implementação, pois combina estratégias simples, como: a heurística construtiva aleatória e gulosa, a busca local, que como visto, pode ser implementada com variadas estratégias e o uso da lista restrita de candidatos (*LRC*), que permite reduzir o escopo das variáveis consideradas na heurística construtiva.

Uma das vantagens do GRASP é a sua flexibilidade na busca local e na composição da *LRC*, pois, uma meta-heurística flexível viabiliza o tratamento de acordo com o problema que está sendo estudado e diferentes problemas requerem diferentes estratégias, além disso, uma meta-heurística flexível permite a combinação com outras heurísticas ou meta-heurísticas formando os métodos híbridos, que para determinados problemas, podem vir a obter excelentes resultados.

10.7 Exercícios da Aula

1. Execute o programa implementado nesta aula por 10 vezes e obtenha:

   - **Melhor FO** - melhor valor obtido para a **FO** entre os testes executados
   - **FO Média** - média dos valores da função objetivo
   - **Desvio da FO** - desvio entre os valores observados da **FO**
   - **Tempo Médio** - média dos tempos para encontrar a solução com ótimo local entre os testes executados
   - **Melhor tempo** - menor tempo para encontrar a melhor solução entre os testes executados

   Utilize a fórmula a seguir para calcular o desvio da **FO**:

   $$desvio = \frac{FOMédia - MelhorFO}{MelhorFO} \times 100 \tag{10.2}$$

2. Faça um programa em Python para otimizar a alocação de itens na mochila com os dados disponibilizados no início da aula 9. Utilize o GRASP tendo como busca local o **método da melhor melhora**.

3. Repita a bateria de testes executada no primeiro exercício, mas agora considere o programa implementando no segundo exercício. Após obter o resumo dos dados, compare com os resultados obtidos no primeiro exercício. Os resultados foram melhores ajustando a busca local para o **método da melhor melhora**?

4. Ajuste o programa construído para o segundo exercício, mudando a estratégia de busca local para a heurística do **método não ascendente/descendente randômico**.

5. Ajuste o programa construído no exercício anterior, mudando a busca local para o **método da descida em vizinhança variável**.

6. Faça novamente as 10 baterias de teste para os dois últimos programas implementados e por fim, compare os resultados registrados para todos os testes. Qual busca local proporcionou os melhores resultados? Justifique.

AULA 11

Meta-heurística Busca Tabu

Metas da Aula

1. Compreender o modo de funcionamento da meta-heurística Busca Tabu.
2. Implementar a meta-heurística Busca Tabu em linguagem Python para o problema da mochila múltipla (PMM).
3. Realizar testes que permitam variar os parâmetros da Busca Tabu.

Ao término desta aula, você será capaz de:

1. Implementar a meta-heurística Busca Tabu em linguagem Python.
2. Entender como aplicar a meta-heurística Busca Tabu em um problema a partir de um pseudo-código.

11.1 Meta-heurística Busca Tabu

A meta-heurística **Busca Tabu** foi introduzida por Glover (1986). O nome Tabu tem origem da língua Tongan, falada em Tonga, na Polinésia, e refere-se a um fato ou objeto que, por ser sagrado, não pode ser tocado ou mencionado (GOLDBARG; GOLDBARG; LUNA, 2016, p. 86). E de fato, essa é uma das estratégias utilizadas na Busca Tabu para fugir de ótimos locais. Conforme Goldbarg, Goldbarg e Luna (2016, p. 86), a Busca Tabu tem 3 princípios básicos de funcionamento:

Princípios básicos de funcionamento da Busca Tabu

1. Uma busca eficiente não deve revisitar soluções.
2. Manter registro de todas as soluções visitadas é caro computacionalmente, é melhor armazenar alterações nas soluções ou em variáveis.
3. Memorizar alterações nas soluções não elimina a possibilidade de revisitar soluções, apenas minimiza as chances disso acontecer.

Assim, basicamente a Busca Tabu funciona de modo a fazer uma gestão eficiente da memória com o objetivo de evitar que configurações sejam revisitadas. Uma configuração na Busca Tabu refere-se a um arranjo de variáveis que forma uma solução (GLOVER; KOCHENBERGER, 2003). O funcionamento da Busca Tabu se baseia em 4 ideias básicas:

Ideias básicas da Busca Tabu

1. Utilizar uma heurística de descida
2. Mover para o melhor vizinho
3. Criar uma lista tabu
4. Critério de aspiração

11.2 1ª ideia: Utilizar uma heurística de descida

Observe a figura 15, ela ilustra uma situação que pode ocorrer em uma busca em vizinhança. Ao realizar uma busca local a partir de uma solução inicial, representada pelo ponto azul, é possível que a heurística "fique presa"no ótimo local, representado pelo ponto vermelho, pois ela não conseguirá continuar a sua busca no espaço amostral, visto que ao visitar os "próximos vizinhos", todas as soluções terão "pior"FO, impossibilitando que a heurística aceite tais soluções. Tal situação está relacionada à segunda ideia.

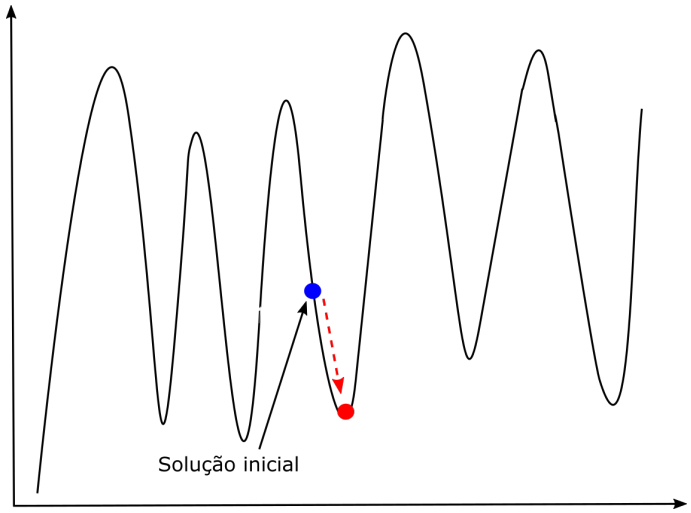

Figura 15 – Exemplo em que a heurística fica "presa"no ótimo local

11.3 2ª ideia: mover para o melhor vizinho

No exemplo da figura 16, a heurística, para não ficar presa no ótimo local, seleciona uma solução vizinha mesmo ela sendo "pior"que a solução anterior, isso pode gerar uma situação indesejada de formação de ciclos, ou seja, o retorno a um ótimo local já visitado anteriormente, o que inviabilizará a busca heurística não só em termos de processamento, mas também na eficácia em encontrar a solução desejada. Para evitar a ciclagem a Busca Tabu adota a 3ª ideia.

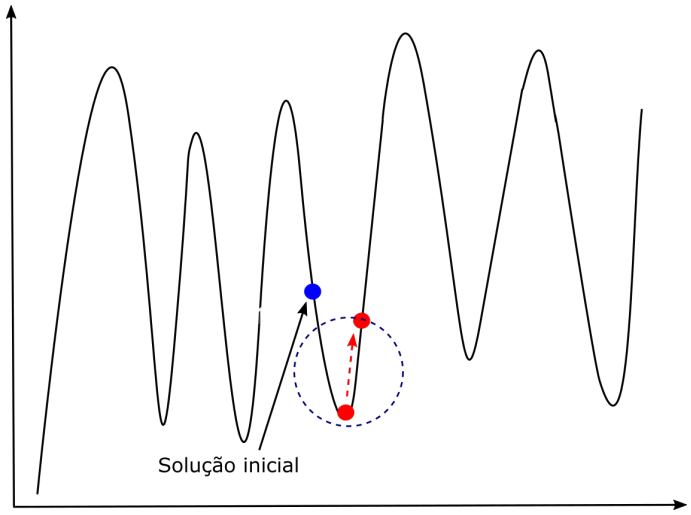

Figura 16 – Exemplo em que a heurística força o movimento para uma solução vizinha

11.4 3ª ideia: Criar uma lista Tabu

Para resolver o problema de ciclagem, a Busca Tabu compõe uma lista de soluções já visitadas, essa é a lista Tabu, em que os elementos que fazem parte dela, se tornam

em um primeiro momento "proibidos", permitindo assim evitar a ciclagem, conforme exemplificado na figura 17, em que o ponto azul, que já havia sido visitado, não será mais examinado, pois agora ele faz parte da lista Tabu. Um problema neste caso, é que armazenar todas as soluções já visitadas é computacionalmente inviável, pois tem um alto custo de memória, assim, em geral adota-se a estratégia de armazenar $|T|$ soluções visitadas, o que leva a evitar a ciclagem até o limite de $|T|$ iterações. Determinar esse $|T|$ também é um problemas a ser resolvido na busca Tabu, pois um valor de $|T|$ muito pequeno evitará ciclos em poucas iterações e um $|T|$ muito grande requer grande espaço de memória (GOLDBARG; GOLDBARG; LUNA, 2016, p. 87).

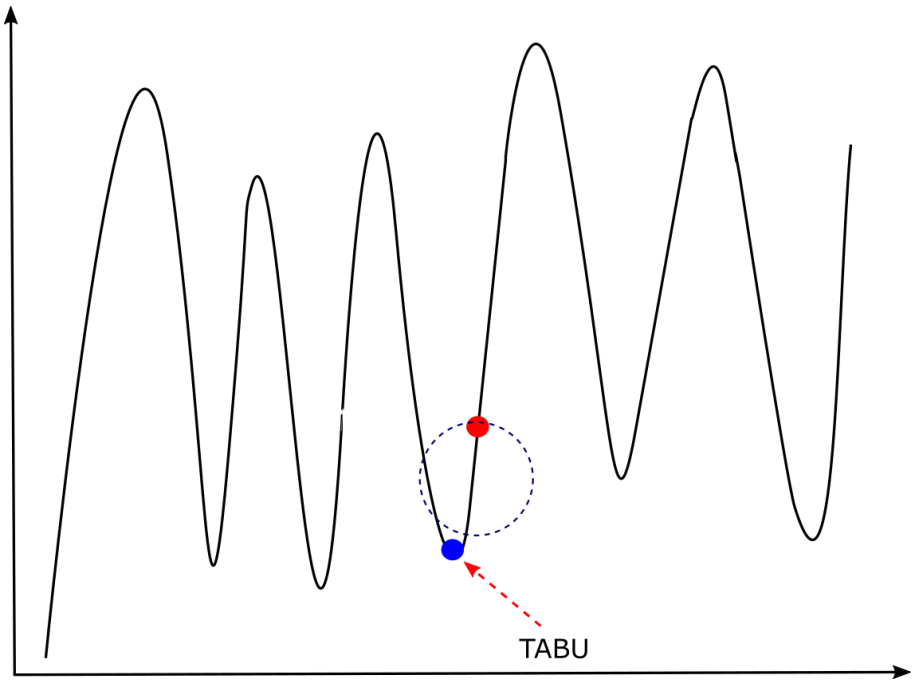

Figura 17 – Exemplo em que um movimento se torna proibido por fazer parte da lista Tabu

Para minimizar o consumo de memória e aumentar assim o potencial de adotar um maior valor para $|T|$, é comum adotar o uso de uma estrutura que armazene as características das variáveis ou movimentos que representam as soluções já visitadas ao invés de armazenar a configuração, ou seja a solução, contudo, ocorre um outro problema, ao controlarmos os movimentos e não as configurações, é possível que diferentes configurações sejam classificados como iguais, em termos de movimentos. Para resolver este problema entra em cena a 4ª ideia.

11.5 4ª ideia: Critério de aspiração

A aspiração é o processo em que será retirado um movimento da lista Tabu com dois objetivos, primeiro, evitar o problema de configurações distintas serem consideradas iguais, segundo garantir a melhoria na solução (GOLDBARG; GOLDBARG; LUNA, 2016, p. 89). O critério é a forma que será adotada para retirar o status de movimento proibido. Dois possíveis critérios de aspiração são:

- **Aspiração por objetivo:** permitir o movimento proibido pela lista tabu se o valor da função objetivo for melhor que o valor global atual (desde que a nova solução

não tenha sido visitada anteriormente, pois assim, ocorreria a ciclagem) (GLOVER; KOCHENBERGER, 2003, p. 44).

- **Aspiração por default:** realizar o movimento Tabu mais antigo, se todos os possíveis movimentos forem Tabu's.

11.5.1 Exemplo de função de aspiração por objetivo

Considere $A()$ como sendo a função de aspiração, $f()$, como sendo a função de cálculo da *FO*, s como sendo a solução vizinha que está sendo examinada e s^* como sendo a melhor solução encontrada até então. Se s estiver na lista Tabu e se $A(f(s)) = f(s^*)$, então o movimento só será aceito se ele conduzir a um vizinho melhor que s^*.

Esse critério considera o fato de que soluções melhores que a solução s^* corrente, ainda que geradas por movimentos tabu, não foram visitadas anteriormente. Isto ocorre porque, como já falado, movimentos tabu podem impedir o retorno a soluções ainda não geradas.

11.6 Algoritmo Base da Busca Tabu

```
1  procedimento BuscaTabu(A, BTmax, |T| e S)
2  S* ← S
3  Iter ← 0
4  MelhorIter ← 0
5  T ← ∅
6  inicializar(A)
7  enquanto (Iter - MelhorIter ≤ BTmax) faca
8      Iter ← Iter + 1
9      gerar(S' ← S ⊕ m-> S' ∈ V ⊆ N(S) e m ∉ T ou f(S') < A(f(S)))
10     atualizar(T)
11     S ← S'
12     Se (f(S') < f(S*)) entao
13         S* ← S
14         MelhorIter ← Iter
15     fimse
16     atualizar(A)
17 fimenquanto
18 retorna S*
```

Como pode-se observar no código base, similar a outras heurísticas, a Busca Tabu começa a partir de uma solução inicial, representada por S, que pode, por exemplo, ser gerada por uma heurística construtiva aleatória. A partir do início, em cada iteração a Busca Tabu irá explorar um subconjunto da vizinhança da solução corrente, entre as linhas 7 e 17 do código base. A solução nessa região de vizinhança com melhor valor torna-se a solução corrente, conforme a linha 11, mesmo que, o seu valor de *FO* seja pior que a atual solução corrente, conforme pode-se verificar na linha 9, pelo trecho: $f(S') < A(f(S))$. A escolha do melhor vizinho, faz parte da estratégia para escapar de um ótimo local. Contudo, essa ação pode levar o algoritmo à ciclagem, ou seja, voltar a uma solução já gerada anteriormente. Neste caso, a lista Tabu vai minimizar esta ocorrência, pois nela estão registrados os movimentos proibidos, conforme a linha 9, no trecho: $m \notin T$.

A lista tabu clássica contém os movimentos reversos aos últimos $|T|$ movimentos realizados, em que $|T|$ é um parâmetro do método, conforme a linha 1 do código base,

e funciona como uma fila de tamanho fixo, isto é, quando um novo movimento é adicionado à lista, o mais antigo sai, sendo este procedimento realizado na linha 10. Como já mencionado, A lista tabu reduz o risco de ciclagem garantindo o não retorno, por $|T|$ iterações, a uma solução já visitada anteriormente. Mas, também pode proibir movimentos para soluções que ainda não foram visitadas. Neste caso, a função de aspiração é um mecanismo que retira, sob certas circunstâncias, o status tabu de um movimento.

11.7 Implementando a meta-heurística Busca Tabu para o PMM

Antes de implementar a Busca Tabu para qualquer problema, é preciso definir a estrutura que será utilizada no registro da lista Tabu. Como já mencionado, armazenar todas as soluções visitadas é inviável, de forma que, foi adotado o critério de armazenar $|T|$ soluções visitadas. Além disso, a estrutura de dados escolhida segue conforme apresentado na figura 18. Observe que a estrutura é similar a uma matriz, em que, na primeira linha serão registrados os objetos e na segunda linha serão registradas as mochilas onde o objeto foi armazenado, assim, essa matriz terá a primeira dimensão fixa e a segunda dimensão vai sofrer variação no tamanho conforme o parâmetro $|T|$ da busca Tabu.

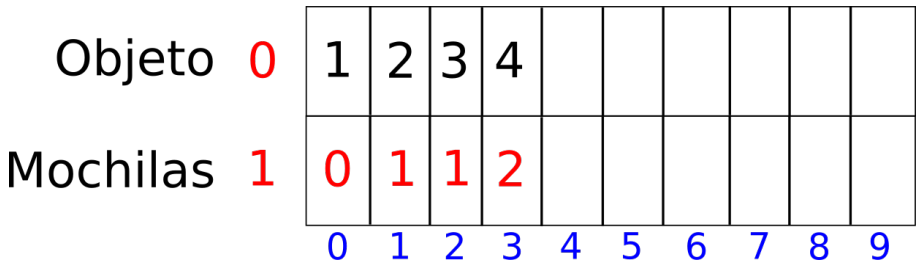

Figura 18 – Estrutura de Dados para a lista Tabu do PMM

Similar ao **GRASP** apresentado na aula 10, a busca Tabu fará uso das funções já implementadas na aula 9, como: a função responsável pela leitura dos dados, a função do cálculo da *FO* e a heurística construtiva para obter a solução inicial, desta forma, a próxima função que deve ser implementada é a própria busca Tabu. Siga os passos a seguir para implementar esta função:

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Cole o código em um editor de textos plano de sua preferência;
3. Salve o arquivo com o nome: **BuscaTabu.py** na pasta **C:\AulaPython**.

```
1 import time
2 import copy as cpy
3 import numpy as np
4 import calcFO as CF
5
6 def buscaTabu(tempo, T, solIni, numObj, numMoc, vetValObj, vetPesObj, vetCapMoc):
```

```
7     ini = time.time()
8     achouT = time.time()
9     qtd = 0
10    lista = np.zeros([2, T])
11    mSol = solIni
12    #calcula a FO da solucao
13    mFOGlobal = CF.calcFO(mSol, numObj, numMoc, vetValObj, vetPesObj, vetCapMoc)
14    sol = cpy.copy(mSol)
15    while 1:
16        #melhor vizinho
17        mFO = float("-inf") #obtendo valor infinito negativo
18
19        mO = -1
20        mM = -1
21        for i in range(numObj):
22            moc = sol[i] #guarda a mochila atual do objeto
23            for j in range(numMoc):
24                #verificar a posicao da lista Tabu
25                pos = -1
26                aspirou = 0
27                for k in range(T):
28                    if lista[0, k] == i and lista[1, k] == j:
29                        pos = k
30                        break
31
32                if pos == -1:
33                    #a configuracao nao esta na lista tabu
34                    sol[i] = j
35                    FO = CF.calcFO(sol, numObj, numMoc, vetValObj, vetPesObj,
                          vetCapMoc)
36                    if FO > mFO:
37                        mO = i
38                        mM = j
39                        mFO = FO
40                else:
41                    #esta na lista tabu, mas a FO e melhor que a FO Global
42                    sol[i] = j
43                    FO = CF.calcFO(sol, numObj, numMoc, vetValObj, vetPesObj,
                          vetCapMoc)
44                    if FO > mFOGlobal:
45                        #aspiracao por objetivo
46                        mO = i
47                        mM = j
48                        mFO = FO
49                        #salva um flag para nao incluir na lista tabu novamente
50                        aspirou = 1
51                #ajustando para mochila original
52                sol[i] = moc
53
54        #atualizando a lista Tabu
55        if mO != -1:
56            #algum vizinho foi aceito
57            sol[mO] = mM #o objeto na posicao aceita recebe a melhor mochila
58            FO = mFO #atualiza a FO
59            if (aspirou == 0): #nao usou o criterio de aspiracao
```

```
60                  lista[0, qtd] = mO
61                  lista[1, qtd] = mM
62                  qtd += 1
63                  if qtd >= T:
64                      qtd = 0
65          else:
66              #nenhum vizinho aceito
67              sol[lista[0,0]] = lista[1,0] #realiza a aspiracao por default
68              FO = CF.calcFO(sol, numObj, numMoc, vetValObj, vetPesObj, vetCapMoc)
69
70          if FO > mFOGlobal:
71              mSol = cpy.copy(sol)
72              achouT = time.time()
73
74          fim = time.time()
75          #verifica se deve continuar executando
76          if fim <= (ini + tempo):
77              continue
78          else:
79              break
80
81      return mSol, (achouT - ini)
```

O código-fonte da busca Tabu e relativamente maior do que o código base apresentado, contudo, essa diferença pode variar de problema para problema. Entre as linhas 1 e 4 foi realizada a importação das bibliotecas e funções que serão necessárias. A linha 6 é o inicio da função da busca Tabu, note que ela possui os argumentos *tempo* que possibilita determinar o tempo de execução, e o argumento *T* referente ao tamanho da lista Tabu e os demais argumentos relacionados ao PMM. A possibilidade de poder configurar o tempo e o tamanho da lista Tabu permite calibrar a meta-heurística e assim, obter uma maior eficácia no objetivo. Na função, os primeiros passos são a inicialização das variáveis de tempo, da lista Tabu, da melhor solução, cuja variável é *mSol* e da solução corrente, em que a variável é denominada como *sol*. Os valores para *FO* também são inicializados, a melhor FO global, na linha 13 e a melhor FO corrente, dentro do laço na linha 17, com o valor infinito, pois assim, na primeira iteração a primeira solução vizinha será aceita.

O laço **while** na linha 15 do código, inclui as instruções que farão a busca pelo melhor vizinho, a atualização da lista Tabu e a aceitação ou não da solução vizinha como solução global, este laço possui uma condição que será sempre verdadeira, contudo, o trecho entre as linhas 76 e 79 possibilitará interromper o laço, de forma que, este funcionará na prática, como se fosse um **do-while**. Além disso, nesta implementação, optou-se pela estratégia de armazenar apenas a posição do objeto e da mochila, ao fazer a busca em um vizinho que seja "melhor", assim, as variáveis que serão responsáveis por este controle, foram inicializadas nas linhas 19 e 20, *mO* para armazenar a posição do objeto e *mM* para armazenar a posição da mochila.

O laço iniciado na linha 21 percorre os objetos e várias operações serão executadas ao longo de suas iterações, primeiro, guarda a mochila do objeto analisado, depois inicia um novo laço que irá percorrer as mochilas e para cada mochila avaliada, verificar se aquela solução está na lista Tabu, caso esteja, guarda a posição e interrompe o laço. Após verificar se a solução está na lista Tabu, o trecho entre as linhas 32 e 39, salva as posições da solução, caso a *FO* da solução vizinha seja melhor que a *FO* da solução corrente. Se a solução estiver na lista Tabu, então é feita uma comparação com a FO global, e caso a FO da solução vizinha seja melhor que a FO da solução global, então

as posições são salvas. Este trecho do código equivale à **aspiração por objetivo**, pois mesmo que a solução esteja na lista Tabu, ela será aceita pois tem melhor FO que a solução global. Após percorrer os objetos em busca de um melhor vizinho, a mochila em que, o objeto estava alocado, é novamente atribuída ao objeto, na linha 52.

Entre as linha 55 e 68 do código, é realizada a inclusão da solução aceita na lista Tabu. Observe duas coisas, primeiro, a variável *qtd* é responsável pelo controle da próxima posição vaga na lista, de forma que, quando a lista fica cheia, essa variável é reiniciada, na linha 64, permitindo que novas soluções sejam incluídas e as soluções mais antigas na lista são removidas. O outro detalhe está relacionado à **aspiração por objetivo**, que ocorre quando a solução, apesar de ter sua configuração presente na lista Tabu, será aceita, pois a sua FO é melhor que a FO global, contudo, a solução não pode mais entrar na lista, assim a instrução na linha 59 considera essa questão, permitindo o registro, apenas de soluções que não estão na lista atual.

Entre as linhas 65 e 68 é realizada a **aspiração por default** caso não tenha encontrado nenhum vizinho com melhor *FO*. A aspiração adotada neste código utiliza uma estratégia simples, que podemos chamar de *FIFO*, sigla em inglês para *First In, First Out*, ou seja, primeiro a entrar, primeiro a sair. E de fato, é o que ocorre, veja que a instrução da linha 67 equivale a atribuição da configuração (objeto, mochila) armazenada na primeira posição da lista Tabu à primeira posição da solução. Ou seja, foi feita uma alteração arbitrária na solução, aceitando inclusive a piora, se for o caso, com o objetivo de causar uma perturbação no processamento da meta-heurística. Após isso, é realizado o cálculo da FO para esta nova solução, na linha 68 do código. Por fim, caso a solução vizinha seja melhor que a solução global, condição da linha 70, então ela será aceita como melhor solução atual, conforme a instrução da linha 71 e o tempo para encontrá-la é registrado na variável *achouT*. Entre as linhas 74 e 79 é realizada a verificação da condição do laço **while**, caso o laço tenha que permanecer ativo, então executa a linha 15, caso contrário, avança para a linha 81 e retorna a melhor solução e o tempo para encontrá-la.

Similar ao que foi feito para o **GRASP**, neste caso também será necessário fazer ajustes no programa principal. Realize os passos a seguir para fazer estes ajustes.

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Abra o arquivo **prgMain.py** presente na pasta **C:\AulaPython**;
3. Cole o código substituindo o código atual do arquivo;
4. Salve o arquivo.

```
1  import leDados as LD
2  import calcFO as CF
3  import BuscaTabu as BT
4  import heuConAle as HCA
5
6  numObjetos = numMochilas = 0
7  T = 10 #neste exemplo adotou-se o tamanho de 10 a lista Tabu
8  tempoExec = 10 #neste exemplo adotou-se 10 segundos de tempo de execucao
9  vetValoresObjetos = []
10 vetPesosObjetos = []
11 vetCapacidadeMochilas = []
```

```
12 numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas = LD.leDados()
13
14 solIni = HCA.heuConstrutivaAleatoria(numObjetos, numMochilas)
15
16 #obtendo uma solucao pela meta-heuristica Busca Tabu
17 sol, tempo = BT.buscaTabu(tempoExec, T, solIni, numObjetos, numMochilas,
       vetValoresObjetos, vetPesosObjetos, vetCapacidadeMochilas)
18
19 #calculando a FO da solucao obtida
20 FO = CF.calcFO(sol, numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas)
21
22 print "Solucao: ", sol
23 print "FO: ", FO
24 print "Achou em: %.2f " % tempo
```

O código-fonte do programa principal teve poucas alterações, a primeira delas, na linha 3, é a importação do arquivo de código-fonte da busca Tabu, já implementado. Na linha 7 foi declarada a variável *T*, responsável pela configuração do tamanho da lista, neste exemplo, foi utilizado o tamanho 10, conforme a linha 7 do código-fonte. Na linha 17 também foi realizado um ajuste, a invocação da função **buscaTabu()**, observe que, ao invocá-la, são informados os parâmetros *tempoExec*, *T* e *solIni*, e as dados referentes ao PMM. O restante do código não foi alterado. Para testar a meta-heurística **Busca Tabu**, abra o interpretador do Python e execute o programa principal, conforme o trecho a seguir.

```
1 >>> runfile('C:/AulaPython/prgMain.py', wdir='C:/AulaPython')
2 Solucao:  [1, 3, 2, 2, 3, 3, 0, 0, 1, 1, 2, 0, 3, 1, 2, 2, 3, 3, 3, 2]
3 FO:  9999
4 Achou em: 10.02
```

Como não foram realizadas alterações na estrutura geral do programa principal, então o formato de apresentação dos resultados segue o mesmo padrão dos testes anteriores, apresentando a melhor solução encontrada, conforme a linha 2 do exemplo, a *FO* da solução, linha 3 e o tempo para encontrar a solução na linha 4. Para o teste deste livro, a melhor solução tem *FO* igual à **9.999** e foi encontrada aos **10.02** segundos da execução da meta-heurística, isso mostra que, especificamente neste teste, a busca Tabu encontrou a melhor solução já no final de sua execução, pois o tempo configurado para a execução é de 10 segundos.

11.8 Calibração dos parâmetros

O parâmetro $|T|$ é o tamanho da lista tabu, como já mencionado, o tamanho da lista pode influenciar negativamente ou positivamente, tanto no desempenho da Busca Tabu, quanto na eficácia em encontrar a solução. Uma opção é definir o tamanho da lista tabu dinamicamente, por um intervalo $[t_{min}, t_{max}]$, que deve ser mudado periodicamente, exemplo: a cada $2t_{max}$ iterações, ou ainda, aumentar o tamanho da lista enquanto estiver na região plana[1] e retornar ao tamanho original quando houver mudança no

[1] Região plana refere-se à uma determinada área do espaço de soluções que está sendo percorrido, mas não são encontradas soluções com valor de função melhor ou refere-se à área em que são obtidas diferentes soluções com *FO* iguais.

valor da função de avaliação, assim, pode-se minimizar as chances de uma solução já visitada ser novamente avaliada.

O parâmetro A é a função de aspiração. O parâmetro $BTmax$ é o número máximo de iterações sem melhora no valor da melhor solução. O critério de parada também pode ser ativado quando o valor da melhor solução chega a um valor já conhecido, o ótimo, por exemplo. Por fim, o parâmetro S é a solução inicial, que pode ser obtida por uma heurística construtiva aleatória ou gulosa. Assim, como em outras heurísticas, esses parâmetros irão variar muito de problema para problema, de forma que, a melhor forma de calibrar é empiricamente, ou seja, por testes e análise dos resultados.

11.9 Resumo da Aula

Como visto ao longo da aula, a busca Tabu é uma meta-heurística que faz uso de três princípios básicos: [1] adotar uma estratégia para evitar revisitar soluções, [2] registrar apenas $|T|$ soluções já visitadas para evitar o custo computacional alto, e uma vez que, foram registradas apenas $|T|$ soluções, no geral, [3] não há garantias de que uma solução não será revisitada. Foi falado também sobre o problema que pode ocorrer ao utilizar uma heurística de descida, no caso, a busca "ficar presa"em um ótimo local, também falou-se sobre a estratégia adotada para fugir destes "ótimos locais", o que poderá levar a outro problema, a ciclagem, ou seja, retornar ao ótimo local anterior, e para evitar movimentos cíclicos, adota-se a lista Tabu, como uma lista de movimentos "proibidos", até certo ponto, pois dada alguma circunstância, um movimento de uma configuração presente nesta lista, poderá ser aceito por meio de aspiração. Assim, falou-se sobre a a **aspiração por objetivo** e a **aspiração por default**.

Por fim, apresentou-se o código base da Busca Tabu, implementou-se a mesma em linguagem Python e discutiu-se brevemente sobre a calibração dos parâmetros. Basicamente os principais parâmetros são o tamanho da lista tabu, geralmente representado por T e o parâmetro de número de iterações da busca. Ambos os parâmetros podem influenciar na eficácia da busca e existem estratégias diferentes para configurá-los. Por exemplo, o número de iterações pode ser substituído por um tempo em segundos, minutos, etc. e o tamanho da lista também possui variações em sua configuração. Outros parâmetros, como a função de aspiração e a solução inicial, não são obrigatórios, pois podem ser tratados no próprio código da busca Tabu.

11.10 Exercícios da Aula

1. Execute o programa implementado nesta aula por 10 vezes e obtenha:

   - **Melhor FO** - melhor valor obtido para a **FO** entre os testes executados
   - **FO Média** - média dos valores da função objetivo
   - **Desvio da FO** - desvio entre os valores observados da **FO**
   - **Tempo Médio** - média dos tempos para encontrar a solução com ótimo local entre os testes executados
   - **Melhor tempo** - menor tempo para encontrar a melhor solução entre os testes executados

   Utilize a fórmula a seguir para calcular o desvio da **FO**:

   $$desvio = \frac{FOMédia - MelhorFO}{MelhorFO} \times 100 \tag{11.1}$$

2. Faça um programa em Python para otimizar a alocação de itens na mochila com os dados disponibilizados no início da aula 9. Utilize a **Busca Tabu** tendo como solução inicial a **heurística construtiva gulosa**.

3. Repita a bateria de testes executada no primeiro exercício, mas agora considere o programa implementando no segundo exercício. Após obter o resumo dos dados, compare com os resultados obtidos no primeiro exercício. Os resultados foram melhores ajustando a forma de obter a solução inicial?

4. Ajuste o programa construído para o segundo exercício, mudando o tamanho da lista Tabu para 20.

5. Ajuste o programa construído no exercício anterior, mudando a solução inicial para ser gerada a partir da **heurística construtiva aleatória**.

6. Faça novamente as 10 baterias de teste para os dois últimos programas implementados e por fim, compare os resultados registrados para todos os testes. Qual teste proporcionou os melhores resultados? Justifique.

AULA 12

Meta-heurística Simulated Annealing

Metas da Aula

1. Compreender o modo de funcionamento da meta-heurística Simulated Annealing.
2. Implementar a meta-heurística Simulated Annealing em linguagem Python para o problema da mochila múltipla (PMM).
3. Realizar testes da meta-heurística Simulated Annealing em conjunto com heurísticas de refinamento.

Ao término desta aula, você será capaz de:

1. Implementar a meta-heurística Simulated Annealing em linguagem Python.
2. Entender como aplicar a meta-heurística Simulated Annealing em um problema a partir de um pseudo-código.

12.1 Meta-heurística Simulated Annealing

O **Simulated Annealing** (**SA**) foi proposto por Kirkpatrick, Gelatt e Vecchi (1983), é um dos muitos algoritmos surgidos na computação natural, que é a ciência que faz uso de sistemas naturais como inspiração para resolver problemas computacionais. No caso do **SA**, a inspiração tem origem na área físico-química, em que, busca mimetizar o recozimento de materiais como metais ou vidros (GOLDBARG; GOLDBARG; LUNA, 2016, p. 105).

12.2 Estratégia do Simulated Annealing

De acordo com Goldbarg, Goldbarg e Luna (2016, p. 106) o **SA** baseia a sua estratégia principalmente em três características:

Princípios características do SA

1. Uso de uma solução inicial;
2. Geração aleatória de novas soluções com base na solução corrente;
3. Critério progressivamente elitista para trocar a solução aleatória pela atual.

O fenômeno de recozimento, *annealing*, que serviu de inspiração para o **SA** pertence à termodinâmica. Descrevendo de uma forma simplificada, o recozimento eleva a temperatura do metal injetando energia em sua estrutura cristalina para posteriormente resfriá-lo lentamente. O processo de resfriar o metal lentamente permite que sua reestruturação seja compatível com o estágio de energia associado à sua temperatura(GONZALEZ, 2007, p. 407). Assim, a ideia geral da meta-heurística é solucionar um problema de otimização com o esquema de aquecimento / resfriamento do *annealing*.

No Simulated Annealing a temperatura é o principal parâmetro de controle da busca, equilibrando o esforço computacional entre a intensificação e a diversificação, ou seja, entre a busca local e o espaço de busca considerado. Ao percorrer o espaço de busca, sempre que uma solução melhor que a atual é encontrada, ela será aceita, contudo, faz parte da estratégia do método utilizar um critério progressivo elitista que permite "transferir o foco da busca", em outras palavras, aceitar uma solução pior que a anterior, para fugir de ótimos locais. Assim, ocorre uma analogia como o processo de *annealing*, em que, quanto maior a temperatura, maior será a chance de aceitar uma mudança de estratégia no caminho da busca (GOLDBARG; GOLDBARG; LUNA, 2016, p. 109).

O **SA** inicia com uma solução qualquer e partir dessa solução o procedimento principal gera uma ou mais soluções vizinhas, cada uma dessas soluções equivalem aos "estados possíveis de um metal". Assim, a cada iteração uma das soluções vizinhas é escolhida para se tornar a nova solução corrente, ou seja, analogamente "o estado atual do metal". A aceitação da solução depende da "variação da energia", que equivale ao valor da função objetivo no problema de otimização, de forma que, se o problema é de minimização, busca-se a energia mínima, e o oposto, se o problema for de maximização (GOLDBARG; GOLDBARG; LUNA, 2016, p. 109).

Se o novo estado da energia é menor que o atual estado, ou seja, melhor que o atual, então o novo estado é aceito e passa a ser o estado corrente, mas caso o novo estado da energia seja maior que o atual estado, ou seja, pior, então é feita uma verificação da variação da energia, sendo Δ (variação do custo - *FO*) unidades, em que, a probabilidade de se mudar para o outro estado, mesmo sendo pior, é: $e^{-\Delta/T}$, sendo T a temperatura corrente. Este procedimento é repetido até atingir o equilíbrio térmico.

Com respeito às soluções, estados, cuja energia é maior, ou seja, com *FO* pior que a corrente, a probabilidades de serem trocadas pela solução corrente é maior quando temperatura está mais alta, a medida que a temperatura diminui, a chance de serem aceitas reduz, de forma que, em baixas temperaturas somente estados com pouca energia (soluções de qualidade), tem boas chances de serem trocados pelo corrente. Assim, observa-se que, em temperaturas altas há uma maior diversificação na busca realizada pelo método, o que permite escapar de ótimos locais, e a medida que a temperatura diminui, a diversificação vai sendo trocada pela intensificação (GLOVER; KOCHENBERGER, 2003, p. 303). Por fim, o método termina quando a temperatura se aproxima de zero.

12.3 Algoritmo Base do Simulated Annealing

```
1  procedimento simulatedAnnealing(α, SAmax, T0, Tc e S)
2  S* ← S
3  IterT ← 0
4  T ← T0
5  enquanto (T > Tc) faca
6      enquanto (IterT < SAmax) faca
7          IterT ← IterT + 1
8          gerar(um vizinho S' ∈ N(S))
9          Δ ← f(S') − f(S)
10         se (Δ < 0) entao
11             S ← S'
12             se (f(S') < f(S*)) entao
13                 S* ← S'
14             fimse
15         senao
16             tomar(x ∈ [0,1])
17             se (x < e^(−Δ/T)) entao
18                 S ← S'
19             fimse
20         fimse
21     fimenquanto
22     T ← α * T
23     IterT ← 0
24 fimenquanto
25 retorna S*
```

O **SA** é provavelmente uma das meta-heurísticas mais fáceis de implementar, como pode-se ver na simplicidade do código base. O procedimento recebe como argumentos de entrada a taxa de resfriamento, representada pelo α, o número de iterações com o argumento *SAmax*, a temperatura inicial T_0, a temperatura de congelamento representada por T_c e a solução inicial S. Os primeiros passos do algoritmo são a inicialização da solução global com a solução inicial, na linha 2, a inicialização da variável *IterT*, res-

ponsável pela contabilização das iterações e a inicialização da temperatura de controle, na linha 4.

O primeiro laço, da linha 5 do código, faz o controle da temperatura, com a condição $T > T_c$, ou seja, enquanto a temperatura armazenada na variável T for maior que a temperatura de congelamento, da variável T_c, o laço continuará executando as iterações. A temperatura de congelamento é fixa, ou seja, ela não se altera ao longo da execução do algoritmo, mas a temperatura inicial atribuída a T na linha 4, vai diminuir a cada iteração de acordo com a taxa de resfriamento, como pode-se observar na linha 22, em que $T \leftarrow \alpha * T$, ou seja, suponha que a temperatura inicial seja **100** e que a taxa de resfriamento seja **0,95**, então ao executar essa instrução pela primeira vez, a temperatura vai cair para **95**, na iteração seguinte vai cair para **90,25** e assim por diante até que a temperatura fique menor que a temperatura de congelamento, provocando assim o interrompimento do primeiro laço, e o método terminará executando a linha 25 que retorna a solução encontrada.

Analisando agora o código executado dentro do laço externo (primeiro laço), observa-se outro laço na linha 6, este é responsável pelas iterações que farão a busca pelo estado em que será atingido o equilíbrio térmico. Observe que a condição para a sua manutenção é **IterT < SAmax**, ou seja, *IterT* inicia com zero, conforme a linha 3 e será incrementado a cada iteração, conforme a linha 7, até atingir o valor definido em *SAmax*, concluindo assim o laço. Contudo este bloco de instruções se repetirá a cada iteração executada no laço externo da linha 5, pois a variável *IterT* é reinicializada em zero na linha 23, após a conclusão do laço interno da linha 6. O objetivo de se repetir o laço interno em cada iteração do laço externo, é que, para cada temperatura avaliada no laço externo, é necessário atingir o equilíbrio térmico.

Para atingir o equilíbrio térmico, primeiro gera-se um vizinho S', conforme a linha 8. De posse do vizinho é calculada a variação por meio da fórmula $\Delta \leftarrow f(S') - f(S)$, que equivale a realizar a subtração da *FO* da solução vizinha, S', pela solução corrente, S. Se o valor da *FO* da solução vizinha for maior, então a variação Δ será positiva, se for menor, o contrário. Para um problema de minimização se a variação é positiva, quer dizer que a solução é pior, se for negativa, então a solução é melhor. Agora, observe a linha seguinte, há uma condição, **se (Δ < 0) entao**, como já mencionado, se a variação é negativa, então melhorou, para um problema de minimização, neste caso a solução S' é aceita, conforme a linha 11, na sequência, verifica-se se ela também é melhor que a solução global, S^*, se for, então ela é atualizada conforme a linha 13. Mas, e se a variação for positiva, neste caso a solução piorou, mas ainda é possível que ela seja aceita, caso a condição da linha 17 seja verdadeira, então a solução corrente será trocada pela solução vizinha, na linha 18, mesmo ela sendo pior, isso permitirá que o **SA** consiga escapar de ótimos locais e como já foi destacado, as chances de "soluções piores"serem aceitas nesta condição, é maior em temperaturas altas e vai diminuindo a medida que a temperatura cai, isso pode ser confirmado observando que a temperatura, T, é um fator de ponderação na formula presente na condição da linha 17.

No parágrafo anterior mencionou-se os casos da condição de aceitação relativos ao problema de minimização, em que, a variação, Δ, negativa equivale a uma solução "melhor"e a variação positiva, equivale a uma solução "pior". Mas, e se o problema for de maximização? Então, é possível trocar a instrução da linha 9, por: $\Delta \leftarrow f(S) - f(S')$, para assim, manter a condição de que uma variação positiva equivale a uma solução "pior"e uma variação negativa equivale a uma solução "melhor". Neste caso, deve ser trocado também o sinal que estabelece a relação na condição da linha 12, assim: ($f(S')$ > $f(S^*)$). Ao fazer estas duas alterações, o **SA** estará preparado para um problema de maximização.

12.4 Implementando a meta-heurística SA para o PMM

Assim como foi feito para as outras meta-heurísticas, o **SA** também será implementado para o PMM. E, como nos casos anteriores, o código-fonte da importação dos dados e do cálculo da *FO* poderão ser reutilizados. O código-fonte do programa principal vai precisar apenas de alguns pequenos ajustes. O Simulated Annealing, em sua primeira versão, não faz uso de nenhuma outra função acessória, como uma busca local, de forma que, será preciso implementar apenas a meta-heurística. Para isso, siga os passos:

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Cole o código em um editor de textos de sua preferência;
3. Salve o arquivo com o nome: **SimulatedAnnealing.py** na pasta **C:\AulaPython**.

```
1  import time
2  import math as mat
3  import copy as cpy
4  import random as rand
5  import calcFO as CF
6
7  #gerando a semente
8  rand.seed()
9
10 def SimulatedAnnealing(samax, ti, tc, tx, solIni, numObj, numMoc, vetValObj,
       vetPesObj, vetCapMoc):
11     ini = time.time()
12     achouT = time.time()
13     #guarda a solucao inicial como a melhor
14     mSol = solIni
15     mFO = CF.calcFO(mSol, numObj, numMoc, vetValObj, vetPesObj, vetCapMoc)
16     #atribui a melhor solucao a sol atual
17     sol = cpy.copy(mSol)
18     FO = CF.calcFO(sol, numObj, numMoc, vetValObj, vetPesObj, vetCapMoc)
19     #inicializa a temperatura
20     temp = ti
21     while temp > tc:
22         #enquanto a temperatura inicial e maior que a temp de congelamento
23         for i in range(samax):
24             #solucao vizinha inicia a partir da solucao atual
25             vSol = cpy.copy(sol)
26             #sorteia uma solucao vizinha
27             vSol[rand.randint(0, numObj-1)] = rand.randint(0, numMoc)
28             #calcula a FO da solucao vizinha
29             vFO = CF.calcFO(vSol, numObj, numMoc, vetValObj, vetPesObj, vetCapMoc
                   )
30             #calculando a variacao
31             delta = FO - vFO
32
33             if delta < 0:
34                 #se a variacao e negativa entao aceita
```

```
35                  sol = cpy.copy(vSol)
36                  if vFO > mFO:
37                      #se a FO da vizinha e melhor que a FO da FO global aceita
38                      mSol = cpy.copy(vSol)
39                      achouT = time.time()
40              else:
41                  if rand.random() < mat.exp(-(FO - vFO) / temp):
42                      #se nao e melhor mas aceita piorar
43                      sol = cpy.copy(vSol)
44
45          #resfria a temperatura
46          temp *= tx
47
48      return mSol, (achouT - ini)
```

Entre as linhas 1 e 5 foi realizada a importação das bibliotecas necessárias para a meta-heurística. Na linha 8 foi gerada a semente aleatória, conforme as orientações da aula 7. Na linha 10 inicia a função, cujos argumentos de entrada são: *samax* referente ao número de iterações, *ti* para temperatura inicial, *tc* para temperatura de congelamento, *tx* para taxa de resfriamento, *solIni* para solução inicial e os demais argumentos do PMM. As variáveis são inicializadas entre as linhas 11 e 20, como a inicialização da melhor solução global na linha 14, da solução corrente na linha 17 e da temperatura corrente na linha 20. O laço externo inicia na linha 21, observe que a condição de manutenção do laço é exatamente igual ao do código base já visto e para garantir que o laço termine quando a temperatura se aproximar da temperatura de congelamento, foi incluída a instrução da linha 46 que é responsável por resfriar a temperatura corrente, com base na taxa de resfriamento.

O laço interno na linha 23 vai garantir as iterações que irão buscar o equilíbrio térmico para cada temperatura avaliada no laço externo. Na linha 25 é inicializada a solução vizinha com a solução corrente, na linha 27 são realizados dois sorteios, o primeiro do objeto, ou seja, aleatoriamente é escolhido um objeto para trocá-lo de mochila, que é o segundo sorteio. Após realizar o sorteio, é feito o cálculo da *FO* da solução vizinha, conforme a instrução na linha 29, como já temos a *FO* da solução corrente, então é realizado o cálculo da variação na linha 31. Na linha 33 é verificado se a variação é negativa, pois se essa condição for verdadeira, então a solução será aceita na instrução da linha 35. Se a solução vizinha também for melhor que a solução global, então a solução global é ajustada conforme a linha 38.

Se a variação for positiva ou nula, então temos a situação em que a solução vizinha é pior ou igual, neste caso é realizada a validação da linha 41, seguindo as definições do código base, e em caso verdadeiro a solução será aceita na linha 43. Após finalizar as iterações do laço externo, ou seja, após a temperatura resfriar e atingir a temperatura de congelamento, a solução é retornada ao método que a invocou, conforme a instrução da linha 48. Realize os passos a seguir para testar o **SA** implementado.

Passos

1. Copie o código-fonte disponibilizado a seguir;
2. Abra o arquivo **prgMain.py** presente na pasta **C:\AulaPython**;
3. Cole o código substituindo o código atual do arquivo;
4. Salve o arquivo.

```
1  import leDados as LD
2  import calcFO as CF
3  import SimulatedAnnealing as SA
4  import heuConAle as HCA
5
6  numObjetos = numMochilas = 0
7  vetValoresObjetos = []
8  vetPesosObjetos = []
9  vetCapacidadeMochilas = []
10 numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas = LD.leDados()
11
12 solIni = HCA.heuConstrutivaAleatoria(numObjetos, numMochilas)
13
14 samax = ((numMochilas + 1) * numObjetos) * 1
15 tempInicial = 100
16 tempCongelamento = 0.01
17 txResfriamento = 0.975
18
19 #obtendo uma solucao pela meta-heuristica Simulated annealing
20 sol, tempo = SA.SimulatedAnnealing(samax, tempInicial, tempCongelamento,
       txResfriamento, solIni, numObjetos, numMochilas, vetValoresObjetos,
       vetPesosObjetos, vetCapacidadeMochilas)
21
22 #calculando a FO da solucao obtida
23 FO = CF.calcFO(sol, numObjetos, numMochilas, vetValoresObjetos, vetPesosObjetos,
       vetCapacidadeMochilas)
24
25 print "Solucao: ", sol
26 print "FO: ", FO
27 print "Achou em: %.2f " % tempo
```

Conforme pode ser visto no código-fonte do programa principal, foram realizadas poucas adaptações, a primeira é a linha 3 do código com a importação do arquivo de fonte da função do Simulated Annealing. O segundo ajuste refere-se à configuração dos parâmetros do **SA**, entre as linhas 14 e 17, em que *samax* foi definido conforme o número de mochilas e objetos para possibilitar que este tamanho possa se adaptar conforme os dados. *tempInicial* foi configurado com o valor **100**, *tempCongelamento* foi configurado como **0.01** e *txResfriamento* com **0.975**. Na próxima seção abordamos com mais detalhes sobre a calibração destes parâmetros. O terceiro e último ajuste é a invocação da função responsável por executar a meta-heurística, na linha 20. O restante do código permaneceu inalterado. Agora é possível testar a meta-heurística **SA**, para executar abra o interpretador do Python e execute o programa principal, conforme o trecho a seguir.

```
1 >>> runfile('C:/AulaPython/prgMain.py', wdir='C:/AulaPython')
2 Solucao:  [1, 3, 3, 1, 1, 1, 1, 2, 3, 3, 2, 0, 3, 3, 0, 0, 2, 1, 0, 3]
3 FO:  9418
4 Achou em: 0.24
```

Como pode ser visto, no teste realizado para este livro, a *FO* da "melhor solução"obtida pelo Simulated Annealing, é 9.418 e esta solução foi encontrada em 0.24 segundos. Tudo indica que os parâmetros configurados permitiram uma execução

muito rápida, talvez, por isso, o **SA** não tenha conseguido uma solução com qualidade melhor. A próxima seção tratará sobre a calibração dos parâmetros, que neste caso, poderá melhorar os resultados do **SA**.

12.5 Calibração dos parâmetros

O Simulated Annealing possui 4 parâmetros que irão influenciar no seu desempenho em termos de tempo de processamento e eficácia em obter boas soluções. Além disso, estes parâmetros estão relacionados entre si, de forma que, um pode afetar o outro. A temperatura inicial, representada no código base por T_0, está associada ao número de estados que o Simulated Annealing vai produzir até atingir a temperatura de congelamento, assim, se colocar uma temperatura muito baixa, é provável que o **SA** faça poucas iterações e obtenha poucos estados térmicos, consequentemente. De forma que, para alguns problemas, é possível que temperaturas iniciais baixas não sejam suficientes para produzir boas soluções. Por outro lado, se calibrar o método com uma temperatura muito alta, é provável que o **SA** tenha um desempenho ruim em termos de tempo de processamento, deve-se então equilibrar a temperatura inicial de acordo com o problema. Um entendimento sobre a T_0 é que ela deve ser alta o bastante para permitir movimentos livres entre soluções vizinhas.

Ainda há outro aspecto que deve ser considerado, pois a temperatura inicial é afetada pela temperatura de congelamento, representada no código base pelo T_c, de forma que, não adianta iniciar o **SA** com uma temperatura muito alta e ao mesmo tempo calibrar a temperatura de congelamento, também com um valor muito alto, ou próximo da temperatura inicial, pois assim, a eficácia em produzir estados térmicos, provavelmente será ruim. Desta forma, é muito comum calibrarmos a temperatura de congelamento com valores muito próximos de zero, como: **0,01**, **0,001**, etc. Até mesmo para que se assemelhe à temperatura natural de congelamento. Tendo a temperatura de congelamento sempre como próxima de zero, então o desafio será sempre encontrar a melhor temperatura para inicializar o **SA**.

Como já mencionado, os parâmetros exercem influência uns sobre os outros, e a taxa de resfriamento, representada pelo α, também influencia na temperatura inicial, pois dependendo da taxa de resfriamento a temperatura pode diminuir rápido ou devagar. Este parâmetro deve ser configurado sempre com valores maiores que 0 e menores que 1, ou seja, $(0 < \alpha < 1)$. No geral, se queremos um resfriamento lento, então devemos configurar na faixa $(0.8 < \alpha < 0.99)$, se desejamos um resfriamento mais rápido então podemos adotar a faixa $(0 < \alpha < 0.8)$. Uma sugestão é que este parâmetro seja calibrado com o valor **0.975** ou **0.995**.

O quarto parâmetro, número de iterações, representado por *SAMax*, influencia no equilíbrio térmico que deve ser atingido em cada temperatura que é avaliada pelo **SA**, durante o processamento. Assim, um valor para *SAMax* muito pequeno pode dificultar o **SA** a atingir o equilíbrio térmico, por outro lado, um número muito grande também não traria muitos benefícios, podendo afetar no desempenho, desta forma, é comum que este parâmetro seja calculado com base na dimensão do problema, como foi feito para o PMM, neste livro.

Em resumo, a melhor forma de calibrar os parâmetros é por meio de testes e avaliação dos resultados. Uma boa estratégia é contabilizar e imprimir um resumo de várias informações ao longo dos testes, por exemplo, contabilizar o número de vizinhos gerados e aceitos, se o número de vizinhos aceitos for alto em relação ao número de vizinhos gerados, isso é um bom indicador.

12.6 Reaquecimento / Resfriamento

É comum implementar em algoritmos do Simulated Annealing, regras que possibilitem que ao longo do processamento ele possa ser reaquecido ou resfriado, para melhorar a diversificação quando o número de movimentos consecutivos rejeitados é grande, podendo indicar assim, que o **SA** está ficando "preso"em mínimos locais. Visto que, em temperaturas mais baixas, as chances de novas soluções serem aceitas diminui, assim, parece ser importante executar mais iterações de *SAmax*, quando a temperatura inicial estiver se aproximando da temperatura de congelamento (GOLDBARG; GOLDBARG; LUNA, 2016, p. 110). O esquema de esfriamento utilizado no **SA**, implementado nesta aula, adota um decréscimo geométrico conforme proposto por Kirkpatrick, Gelatt e Vecchi (1983) e Wilhelm e Ward (1987). Observe na figura 19 que o esfriamento neste esquema é atenuado a medida que se aproxima da taxa de congelamento.

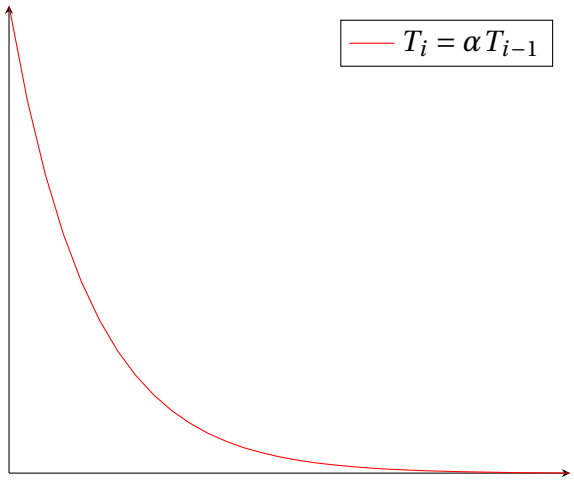

Figura 19 – Esquema de esfriamento com decréscimo geométrico

Há vários outros esquemas de esfriamento que podem ser adotados, como o decréscimo linear, conforme a equação: $T_i = T_{i-1} - k$, em que k é um valor constante. Ou o esquema proposto por Lundy e Mees (1986), conforme a equação: $T_i = \frac{T_{i-1}}{1+\beta T_{i-1}}$, em que $\beta << T_0$. Ou ainda a proposta de Aarts e Korst (1989), conforme a equação 12.1. Além dessas, há ainda várias outras propostas de esquemas de esfriamento na literatura que podem ser exploradas.

$$T_i = \frac{T_{i-1}}{1 + \frac{\beta}{\sigma(T_{i-1})}} \tag{12.1}$$

$$\beta = \frac{ln(1+\delta)T_{i-1}}{3} \tag{12.2}$$

Em que δ é o parâmetro de distância e $\sigma(T_{i-1})$ é o desvio-padrão dos valores obtidos das soluções aceitas no estágio, cuja a temperatura é T_{i-1} (GOLDBARG; GOLDBARG; LUNA, 2016, p. 111).

A figura 20 apresenta dois esquemas juntos para comparação. O esquema de decréscimo geométrico é representado pela linha vermelha e o esquema proposto por Lundy e Mees (1986) é representado pela linha azul. Ao observar o gráfico da figura

20, percebe-se que, para o mesmo período de tempo (eixo x), em temperaturas mais altas, o esquema geométrico teve queda de temperatura com menos intensidade que o esquema proposto por Lundy e Mees (1986). O contrário, ocorre em temperaturas mais baixas, neste caso, é o esquema de Lundy e Mees (1986) que teve a queda de temperatura mais atenuada. Isso quer dizer que, em temperaturas mais altas, o esquema geométrico promoverá maior diversificação na busca pela vizinhança e, em temperatura mais baixas, é o esquema de Lundy e Mees (1986) que vai obter este efeito.

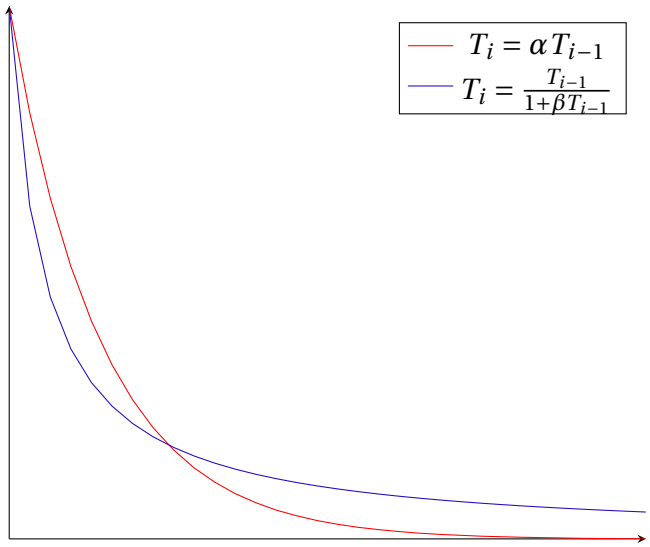

Figura 20 – Comparação do esquema de esfriamento proposto por Lundy e Mees (1986) com o decréscimo geométrico

12.7 Resumo da Aula

Esta aula permitiu entender como é o funcionamento da meta-heurística Simulated Annealing e qual a sua estratégia para obter boas soluções. De uma forma resumida, o SA é uma meta-heurística que, a partir de uma solução inicial, explora a vizinhança, em cada estado térmico, em busca de uma solução com melhor valor de função objetivo. Um dos diferenciais na estratégia do SA, é que dada uma circunstância, ele aceitará uma solução vizinha, mesmo que esta tenha um valor de *FO* "pior". Essa característica favorece a fuga de mínimos locais em temperaturas mais altas e a medida que a temperatura diminui, a probabilidade das soluções "piores"serem aceitas, também diminui, de forma que, há um equilíbrio entre a intensificação e a diversificação na exploração do espaço de busca.

Os principais parâmetros do Simulated Annealing são a temperatura inicial, representada por T_0, a temperatura de congelamento, T_c, a taxa de resfriamento, α, o número de iterações e a solução inicial. Geralmente, a temperatura de congelamento pode ser sempre calibrada com valores próximos de zero (0.01, 0.001, etc.) e a preocupação maior em relação à temperatura deve ser voltada para a temperatura inicial e a taxa de resfriamento, pois ambas irão causar maior influência na eficácia do método. Para a taxa de resfriamento, existem vários esquemas de resfriamento na literatura, como o decréscimo linear, o decréscimo geométrico, o esquema proposto por Lundy e Mees (1986), entre outros. O número de iterações influencia na exploração do espaço de busca do estado térmico corrente, de forma que, um número de iterações muito pequeno, pode dificultar o alcance do equilíbrio térmico, assim, é comum que o número de iterações seja definido com base no problema. Como os parâmetros podem surtir efeitos distintos para variados problemas, a melhor forma de calibrá-los é com testes e avaliação dos testes, ou seja, empiricamente.

12.8 Exercícios da Aula

1. Execute o programa implementado nesta aula por 10 vezes e obtenha:

   - **Melhor FO** - melhor valor obtido para a **FO** entre os testes executados
   - **FO Média** - média dos valores da função objetivo
   - **Desvio da FO** - desvio entre os valores observados da **FO**
   - **Tempo Médio** - média dos tempos para encontrar a solução com ótimo local entre os testes executados
   - **Melhor tempo** - menor tempo para encontrar a melhor solução entre os testes executados

   Utilize a fórmula a seguir para calcular o desvio da **FO**:

   $$desvio = \frac{FOMédia - MelhorFO}{MelhorFO} \times 100 \tag{12.3}$$

2. Ajuste o programa principal de forma a gerar a solução inicial a partir a heurística construtiva gulosa, além disso, altere os parâmetros de entrada do **Simulated Annealing**, conforme: $T_0 = 200$, $\alpha = 0.8$.

3. Repita a bateria de testes executada no primeiro exercício, mas agora considere o programa ajustado no segundo exercício. Após obter o resumo dos dados, compare com os resultados obtidos no primeiro exercício. Os resultados foram melhores ajustando a solução inicial e os parâmetros?

4. Ajuste o programa construído nesta aula, de forma que, o esquema de resfriamento seja pelo decréscimo linear.

5. Ajuste o programa construído no exercício anterior, mudando o esquema de resfriamento conforme a proposta de Lundy e Mees (1986).

6. Faça novamente as 10 baterias de teste para os dois últimos programas implementados e, por fim, compare os resultados registrados para todos os testes. Qual esquema de resfriamento proporcionou os melhores resultados? Justifique.

Referências

AARTS, E.; KORST, J. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. New York, NY, USA: John Wiley & Sons, Inc., 1989. ISBN 0-471-92146-7. Citado na página 171.

ARORA, J. S. *Introduction to Optimum Design*. 2. ed. San Diego: Elsevier Academic Press, 2004. ISBN 0-12-064155-0. Citado na página 109.

BIRD, S.; KLEIN, E.; LOPER, E. *Natural Language Processing with Python*. 1. ed. California: O'Reilly books, 2009. ISBN 978-0-596-51649-9. Citado na página 16.

BORGES, L. E. *Python para Desenvolvedores*. 2. ed. Rio de Janeiro: Edição do Autor, 2010. ISBN 978-85-909451-1-6. Citado 5 vezes nas páginas 10, 19, 21, 26 e 78.

COELHO, F. C. *Computação Científica com Python: Uma introdução à programação para cientistas*. Petrópolis: Edição do Autor, 2007. ISBN 978-85-907346-0-4. Citado 6 vezes nas páginas 14, 15, 17, 26, 28 e 44.

CRUZ, F. *Python: Escreva seus primeiros programas*. 1. ed. São Paulo: Casa do Código, 2017. ISBN 978-85-5519-091-9. Citado 9 vezes nas páginas 19, 20, 26, 29, 44, 45, 78, 89 e 90.

DOWNEY, A.; ELKNER, J.; MEYERS, C. *Aprenda Computação com Python Documentation*. 1. ed. [S.l.: s.n.], 2010. Citado 5 vezes nas páginas 13, 33, 35, 36 e 101.

FARAHANI, R. Z.; HEKMATFAR, M. *Facility Location: Concepts, Models, Algorithms and Case Studies*. 1. ed. New York: Physica-Verlag, 2009. ISBN 978-3-7908-2151-2. Citado na página 109.

FEO, T. A.; RESENDE, M. G. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, v. 8, n. 2, p. 67–71, 1989. ISSN 0167-6377. Citado na página 141.

GLOVER, F. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, v. 13, n. 5, p. 533–549, 1986. ISSN 0305-0548. Applications of Integer Programming. Citado na página 151.

GLOVER, F.; KOCHENBERGER, G. A. *Handbook of metaheurististics*. 1. ed. New York: Kluwer Academic Publishers, 2003. ISBN 0-306-48056-5. Citado 7 vezes nas páginas 108, 137, 141, 147, 151, 154 e 165.

GOLDBARG, M. C.; GOLDBARG, E. G.; LUNA, H. P. L. *Otimização combinatória e meta-heurísticas: algoritmos e aplicações*. 1. ed. Rio de Janeiro: Elsevier, 2016. ISBN 978-85-352-7812-5. Citado 16 vezes nas páginas 100, 108, 109, 110, 111, 115, 117, 129, 136, 137, 141, 147, 151, 153, 164 e 171.

GOLDBARG, M. C.; LUNA, H. P. L. *Otimização combinatória e programação linear: modelos e algoritmos*. 2. ed. Rio de Janeiro: Elsevier, 2005. ISBN 978-85-352-1520-5. Citado na página 108.

GONZALEZ, T. F. *Handbook of approximation algorithms and metaheurististics*. 1. ed. Boca Raton: Chapman & Hall, 2007. ISBN 978-1-58488-550-4. Citado 7 vezes nas páginas 108, 118, 129, 130, 133, 136 e 164.

HART, J.; SHOGAN, A. W. Semi-greedy heuristics: An empirical study. *Operations Research Letters*, v. 6, n. 3, p. 107–114, 1987. ISSN 0167-6377. Citado na página 141.

HERTZ, A.; MITTAZ, M. A variable neighborhood descent algorithm for the undirected capacitated arc routing problem. *Transportation Science*, v. 35, n. 4, p. 425–434, 2001. Citado na página 135.

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. *Science*, v. 220, n. 4598, p. 671–680, 1983. Citado 2 vezes nas páginas 164 e 171.

KUMAR, S. A.; SURESH, N. *Operations Management*. 1. ed. New Delhi: New Age, 2009. ISBN 978-81-224-2883-4. Citado na página 108.

LOPES, A.; GARCIA, G. *Introdução à programação: 500 algoritmos resolvidos*. Rio de Janeiro: Elsevier, 2002. ISBN 85-352-1019-9. Citado 15 vezes nas páginas 23, 29, 30, 35, 36, 39, 45, 50, 51, 52, 53, 59, 72, 75 e 85.

LOPES, H. S.; RODRIGUES, L. C. de A.; STEINER, M. T. A. *Meta-Heurísticas em Pesquisa Operacional*. Curitiba, PR: Omnipax, 2013. ISBN 978-85-64619-11-1. Citado na página 135.

LUNDY, M.; MEES, A. Convergence of an annealing algorithm. *Mathematical Programming*, v. 34, p. 111–124, 1986. Citado 5 vezes nas páginas 1, 171, 172, 173 e 174.

LUTZ, M.; ASCHER, D. *Aprendendo Python*. Porto Alegre: bookman, 2007. ISBN 978-85-7780-013-1. Citado 7 vezes nas páginas 16, 44, 48, 66, 68, 78 e 82.

MCKINNEY, W. *Python for Data Analysis*. 1. ed. Sebastopol: O'Reilly, 2013. ISBN 978-1-449-31979-3. Citado 5 vezes nas páginas 11, 78, 92, 94 e 101.

MLADENOVIć, N.; HANSEN, P. Variable neighborhood search. *Computers & Operations Research*, v. 24, n. 11, p. 1097–1100, 1997. ISSN 0305-0548. Citado na página 135.

WILHELM, M. R.; WARD, T. L. Solving quadratic assignment problems by simulated annealing. *IIE Transactions*, Taylor & Francis, v. 19, n. 1, p. 107–119, 1987. Citado na página 171.


9 788561 890995