UNIVERSIDADE ABERTA DO BRASIL
UNIVERSIDADE FEDERAL DE ALAGOAS
Instituto de Computação
Curso de Bacharelado em Sistemas de Informação

ALGORITMO E ESTRUTURA DE DADOS I

Prof. Ailton Cruz dos Santos

UNIVERSIDADE ABERTA DO BRASIL
UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO

Curso de Bacharelado em Sistemas de Informação
Disciplina:
ALGORITMO E ESTRUTURA DE DADOS I
Prof. Ailton Cruz dos Santos

Texto da disciplina produzido em 2008 e revisado em 2010, 2012 e 2014

Apresentação

Prezado estudante,

Esta disciplina tem um papel importante na sua formação profissional na área de Tecnologia de Informação. O computador é esta sofisticada máquina que a todo momento ganha um novo espaço de aplicação nas atividades humanas. Sabe-se, no entanto, que são os programas nela inseridos os verdadeiros responsáveis pelas suas funcionalidades. Seja bem vindo a esta disciplina onde vocês conhecerão como é possível gerar esses programas e darão os primeiros passos para o exercício de um tipo de atividade que está além daquela de simples usuário. O bastante é seguir o roteiro definido no curso, etapa por etapa, interagindo com o professor, tutores e colegas.

Prof. Ailton Cruz.

Plano de trabalho

TÍTULO DA DISCIPLINA: Algoritmo e Estrutura de Dados I

CARGA HORÁRIA

Presencial: 06 horas; On-line: 114 horas

EMENTA:

História do Computador. Os Computadores e a Resolução de Problemas. Estruturas de Decisão. Vetores e Conjuntos. Cadeia de Caracteres. Subalgoritmos. Recursividade.

OBJETIVOS DA DISCIPLINA

Objetivo geral:

Capacitar o estudante a elaborar soluções de problemas sob a forma de sequências de instruções finitas e objetivas, chamadas de algoritmos, e a concretizar estas soluções na forma de programas de computador.

Objetivos específicos:

O estudante deverá adquirir a habilidade de identificar e combinar as estruturas lógicas de controle e de dados apresentadas durante o estudo dos algoritmos;

Decidir que estruturas de dados são adequadas para a solução de determinados problemas;

Partindo do estudo da estrutura e funcionalidade de uma linguagem de programação, implementar as estruturas identificadas (lógicas e de dados) e testar os programas produzidos.

METODOLOGIA DE ENSINO

Os estudantes deverão assimilar o conteúdo programático a partir da leitura e análise dos textos da disciplina disponíveis no AVA Moodle organizados por semanas. O acesso à bibliografia complementar, a execução de testes dos algoritmos em laboratório de informática, bem como as interações com o professor e tutores usando ferramentas da plataforma Moodle são elementos constitutivos da metodologia desta disciplina.

METODOLOGIA DE AVALIAÇÃO:

A avaliação do estudante acontece de forma contínua mediante análise das interações no AVA Moodle e tarefas semanais. A participação on-line mínima requerida do estudante é sua atividade no fórum sobre os conteúdos da semana e registro em seu blog de suas seções de estudo, da seguinte maneira. No fórum sobre os conteúdos, o estudante deve interagir com os demais participantes a fim de tirar dúvidas, ora as suas, ora as dos colegas. No blog, o estudante deve registrar que roteiro seguiu no seu estudo, suas estratégias, o que conseguiu estudar, que aspectos foram relevantes, a que conclusões chegou, como está sendo seu contato com a tutoria inclusive se fez contato fora do AVA etc. Juntas, as participações no fórum dos conteúdos e registro no blog correspondem a 50% da pontuação da semana. Esta pontuação é completada com uma tarefa sobre o tema da semana. Obs.: Espera-se que as demais participações, como no fórum de discussões gerais, por exemplo, sejam normalmente detalhes daquilo discutido no fórum dos conteúdos ou registrado no blog. Tais participações podem melhorar a pontuação de interação, mas são desconsideradas se não houver conectividade no blog ou no fórum dos conteúdos. A nota do estudante é composta da maneira seguinte:

-Composição da primeira nota: 60% desta é obtida por pontuação das atividades on-line (30% participação em fórum e blog, 30% por tarefas específicas) das cinco primeiras semanas e 40% por prova online individual ao final da quinta semana.

-Composição da segunda nota: 60% desta é obtida por pontuação das atividades on-line das duas últimas semanas e 40% por prova presencial, individual e com consulta, ao final da sétima semana.

CONTRIBUIÇÃO PARA A FORMAÇÃO PROFISSIONAL

Ao final da disciplina, o estudante terá desenvolvido (ou, aprimorado) capacidade para resolver problemas de porte médio por computador, através da combinação de estruturas lógicas e de dados, requisito básico para produção e análise de sistemas de software.

PRÉ-REQUISITOS

Conceitos de Matemática abordados no ensino médio: expressões, trigonometria, geometria, etc.;

Mapas conceituais;

APLICAÇÃO PRÁTICA DA DISCIPLINA

Diante da característica de fundamento da computação, o conhecimento desenvolvido nesta disciplina, aliado a outros específicos ao longo curso, dará ao estudante as condições necessárias para produção nas áreas de redes de computadores, Internet, Web, etc.

CONTEÚDO PROGRAMÁTICO DA DISCIPLINA:

Módulo I - O computador e a resolução de problemas
- 1.1 - O computador e seu modelo lógico
  - 1.1.1 - Histórico
  - 1.1.2 - Hardware e software
- 1.2 - A resolução de problemas por computador
  - 1.2.1 - Do problema ao programa
  - 1.2.2 - O processo de elaboração dos algoritmos

Módulo II - Elementos básicos para elaboração dos algoritmos
- 2.1 - Dados, variáveis e comandos básicos
  - 2.1.1 Tipos de dados
  - 2.1.2 As variáveis e a atribuição de valores
  - 2.1.3 Entrada e saída de dados
- 2.2 – Expressões
  - 2.2.1 Expressões aritméticas
  - 2.2.2 Expressões lógicas;
  - 2.2.3 Expressões literais

Módulo III - Estruturas de controle
- 3.1 - Estruturas de seleção
  - 3.1.1 Formato "se - então"
  - 3.1.2 Formato "se - então - senão"
  - 3.1.3 Formato encadeado
- 3.2 - Estruturas de repetição
  - 3.2.1 A estrutura "para"
  - 3.2.2 A estrutura "enquanto"

Módulo IV - Estruturas de dados
- 4.1 - Estruturas homogêneas básicas
  - 4.1.1 Vetores
  - 4.1.2 Matrizes
- 4.2 - Estruturas homogêneas especiais
  - 4.2.1 Cadeias de Caracteres
  - 4.2.2 Conjuntos.

Módulo V - Modularização de algoritmos
- 5.1 - Subalgoritmos
  - 5.1.1 Definição - parâmetros, retorno
  - 5.1.2 Variáveis locais e variáveis globais
- 5.2 - Recursividade
  - 5.2.1 Definição e aplicações

BIBLIOGRAFIA:

FORBELLONE, A. L. V.; EBERSPACHER, H. F. Lógica de Programação: a Construção de Algoritmos e Estruturas de Dados. 3a ed. São Paulo: Makron Books, 2005. 232 p.

PILGRIM, M. Mergulhando no Python. Rio de Janeiro: Alta Books, 2004

VILARIM, G. Algoritmos: programação para iniciantes. Rio de Janeiro: Ciência Moderna, 2004. 288 p.

FARRER, H. e outros. Algoritmos Estruturados. Rio de Janeiro: Editora Guanabara, 1999. 252 p.

BIBLIOGRAFIA COMPLEMENTAR: A bibliografia complementar é composta de uma coleção dinâmica de links com sugestões de leitura, disponibilizada na página da disciplina no Ambiente Virtual de Aprendizagem (AVA) Moodle.

Plano de tutoria

IDENTIFICAÇÃO DA DISCIPLINA

Disciplina: ALGORITMO E ESTRUTURA DE DADOS I
Carga horária: 120 horas
Professor autor: Ailton Cruz dos Santos
Ementa:
História do Computador. Os Computadores e a Resolução de Problemas. Estruturas de Decisão. Vetores e Conjuntos. Cadeia de Caracteres. Subalgoritmos. Recursividade.

PLANEJAMENTO

MOMENTOS PRESENCIAIS

- PRIMEIRO MOMENTO. Apresentação da disciplina, incluindo sequência do conteúdo, metodologia de ensino, recomendações ao estudante, descrição do processo de avaliação etc. Inclui introdução aos temas da semana corrente e aula em laboratório sobre a linguagem Python se o encontro ocorrer a partir da segunda semana.
- SEGUNDO MOMENTO. Constará de aula de reforço do conteúdo trabalhado até esse momento, avaliação das estratégias adotadas e dos níveis de participação no curso, e introdução aos temas da semana corrente.

REUNIÕES COM A TUTORIA

Para avaliação do andamento da disciplina, reuniões da equipe de professores (professor-coordenador e tutores) deverão ocorrer pelo menos três vezes ao longo do curso:

- A primeira, na semana que antecede o início da disciplina. Pauta: Apresentação do material com o conteúdo, metodologia de ensino e avaliação;
- A segunda, durante a quinta semana. Pauta: Análise do processo ensino-aprendizagem, dos resultados parciais, e discussão de casos particulares;
- Ao final da sétima semana. Pauta: Análise do processo ensino-aprendizagem, da avaliação e dos resultados obtidos;

AVALIAÇÃO DOS ESTUDANTES

Primeira nota da disciplina:

- As atividades realizadas durante as cinco primeiras semanas têm peso de 6,0 pontos. Em cada semana, as interações do estudante no ambiente virtual de aprendizagem (o Moodle), blog e fórum juntas, valem 1,0 ponto, e as atividades dos finais de semana, também. A nota é completada com uma prova individual, online, valendo 4,0 ao final da quinta semana em data a ser marcada.

Segunda nota da disciplina:

- As atividades no Moodle, incluindo aquelas do fim de semana, da sexta e sétima semanas têm peso 6,0. A prova que consta no calendário como **AV2** tem peso 4,0.
- **Critérios para a AV2 -** Será uma prova presencial e individual, envolverá todo conteúdo da disciplina e será com consulta. Os únicos materiais que podem ser consultados são: o texto da disciplina e anotações que o estudante tenha feito durante seus estudos. O material consultado é pessoal. Não é permitido consulta de materiais de colegas durante a prova.

ATIVIDADES A SEREM DESENVOLVIDAS PELOS ESTUDANTES

No início de cada semana:

- Planejar suas atividades da semana, definindo horários e guiando-se pelo tempo mínimo sugerido na página da disciplina no Moodle;

Diariamente:

- Seguir os roteiros de estudo disponíveis no texto da disciplina com teoria e prática;
- Examinar e repetir os exemplos, refletir sobre a teoria, exemplos e laboratórios. Ao concluir esses passos, proceder a sua autoavaliação realizando os exercícios ao final de cada subunidade, fazendo testes dos algoritmos em computador usando a plataforma Python;
- Participar, em todas essas fases acima, do fórum “Sobre os conteúdos”: Interagindo com os colegas e os professores, expondo dúvidas e questionamentos, contribuindo no sentido inclusive de auxílio aos colegas nas resoluções de exercícios, etc. Os

temas que extrapolarem o escopo do curso devem ser levados para o fórum "Discussões Gerais";
- Fazer um registro breve no seu BLOG: Anotar o que conseguiu em cada dia de estudo, que exercícios conseguiu resolver ou não, das suas dificuldades ou observações gerais de deseje fazer sobre os conhecimentos adquiridos;

Ao final de cada semana:
- Resolver um problema simples usando os recursos desenvolvidos na semana respectiva.

ATRIBUIÇÕES DO TUTOR ONLINE
- Tomar ciência das atividades a serem desenvolvidas pelos estudantes com antecedência. Deve estar familiarizado com a *metodologia da disciplina*[1], a plataforma Python e apto a tirar dúvidas dos estudantes e a sugerir novos exercícios;
- Participar regularmente do "Fórum da tutoria" (não visível para os estudantes) onde são tratadas questões operacionais de acompanhamento e avaliação;
- Participar das interações nos fóruns "Discussões Gerais", "Sobre os conteúdos" para contato com seus alunos, como também no "Fórum de notícias";
- Acessar o perfil de cada estudante, acompanhando os registros nos *blog*s e fóruns:
- Conferir a coerência de seus registros, comparando com suas participações nos fóruns;
- Verificar desempenho e a frequência de acesso, alertando em tempo os participantes sobre o encaminhamento de suas atividades;
- Detectar conceitos equivocados tanto nos fóruns quanto nos blogs para a imediata correção;
- Providenciar recursos para auxiliar os estudantes em suas dúvidas;
- Estimular o estudante a fazer os exercícios de autoavaliação e a assumir uma atitude autônoma;
- Corrigir as tarefas ao final de cada semana e atribuir a pontuação em blog e fórum;
- Atribuir a pontuação semanal preferencialmente até a quarta-feira da semana seguinte ou, no máximo, até a quinta-feira (dia em que é aberta a próxima tarefa);
- A pontuação de blog e fórum, em cada um desses itens é atribuída na faixa de 0 a 0,5: fórum é analisado pela qualidade e abrangência por refletir interesse no aprendizado e integração com os colegas e demais participantes do curso; blog é avaliado pela coerência (não quantidade) - deve refletir que o estudante de fato está dando atenção ao curso (estudando, seguindo a metodologia adotada, interagindo,...), enfim, que tem um ***plano individual de estudo*** (mesmo que não escreva explicitamente);
- O fórum central é o "Sobre os conteúdos". Os assuntos que extrapolarem o escopo do curso devem ser levados para "Discussões Gerais". Não serão pontuadas as participações no fórum de discussões gerais ou de notícias, que não revelem consistência com o tema da semana, ou que não tenha sido feita pelo menos uma postagem pertinente no fórum dos conteúdos da semana;
- Manter vigilância principalmente sobre os estudantes que não estão sendo ágeis no registro dos seus blogs (ou estão ausentes no Moodle), enviando-lhes mensagens de motivação, entrando em contato direto se necessário;
- Coordenar a recepção das atividades da semana notificando os estudantes com pendências.

ATRIBUIÇÕES DO TUTOR PRESENCIAL
- Atender os requisitos e a atribuição clássica de suporte às atividades presenciais;
- Participar regularmente do "Fórum da tutoria" (não visível para os estudantes) onde são tratadas questões operacionais de acompanhamento e avaliação;
- Tomar ciência das atividades a serem desenvolvidas pelos estudantes com antecedência. Tirar pessoalmente dúvidas dos estudantes e/ou coordenar grupos de

[1] Obs. O seguinte artigo apresenta os fundamentos da metodologia empregada na disciplina:
SANTOS, A.C. . *Uma Abordagem Integrativa para o Ensino de Algoritmos a Distância*. In: WEI - XXI Workshop sobre Educação em Computação/Congresso da Sociedade Brasileira de Computação, 2013, Maceió. Anais do XXXIII Congresso da Sociedade Brasileira de Computação, 2013. Disponivel em: *http://www.lbd.dcc.ufmg.br/colecoes/wei/2013/0023.pdf*

estudo no polo contando com a participação dos monitores, diagnosticando os casos em parceria com a tutoria online;

- Reservar parte de sua carga horária para um “plantão tira-dúvidas” no polo, em horário a combinar com os estudantes e os monitores (principalmente com respeito a execução de programas);
- Providenciar recursos para auxiliar os estudantes em suas dúvidas;
- Em seu contato com os estudantes, estimulá-los a fazer os exercícios de autoavaliação e a assumir uma atitude autônoma;
- Ajudar a tutoria online na investigação das ausências de estudantes na plataforma combatendo a evasão.

Cronograma/Índice

Introdução

O presente texto é destinado fundamentalmente à preparação dos estudantes que estejam em seu primeiro contato com a tarefa de programação de computadores. Reúne os conteúdos abordados na disciplina de Algoritmo e Estrutura de Dados I.

Os cinco módulos que compõem o material da disciplina possuem parte teórica seguida de exercícios de aprendizagem. O primeiro deles trata dos fundamentos para o uso do computador como ferramenta para a solução de problemas. Passando por questões históricas e princípios de funcionamento, introduz os conceitos de *algoritmo* e *programa*. Onde, em primeira análise, um algoritmo é uma sequência de passos que culminam com a solução de um dado problema e um programa é uma versão do algoritmo compreensível para a máquina.

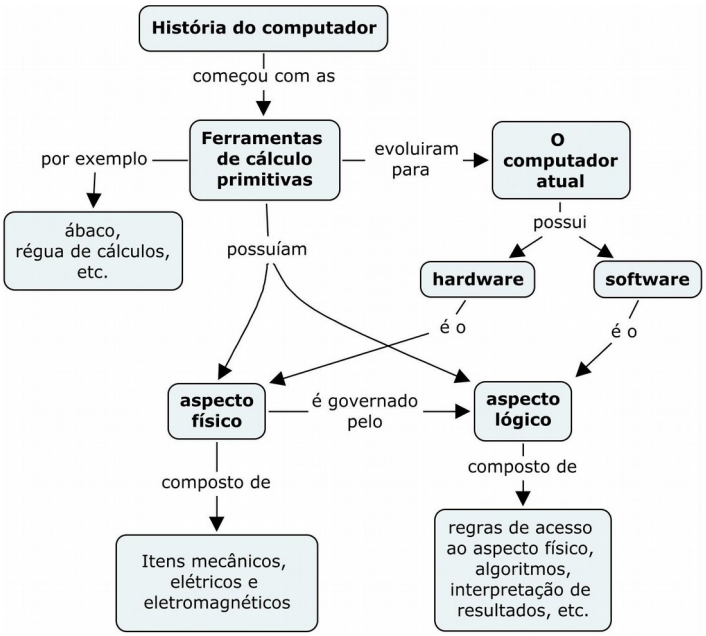
Os demais módulos dizem respeito às técnicas para a efetiva elaboração de algoritmos e programas: Inclui o estudo da lógica de programação e da estruturação dos dados. Nesses módulos, então, a parte teórica é acompanhada de atividades de laboratórios. Os laboratórios são roteiros de elaboração de programas de computador subdivididos em experimentos associados aos conceitos da parte teórica. Esta prática no computador é requisito fundamental para sedimentação dos conceitos.

Os exercícios de aprendizagem colocados após os conteúdos são distribuídos por itens em ordem de complexidade, partindo dos casos constantes na parte teórica e nos laboratórios. A proposta de uso dos recursos busca favorecer o aproveitamento da lógica natural do estudante no aprendizado da lógica de programação.

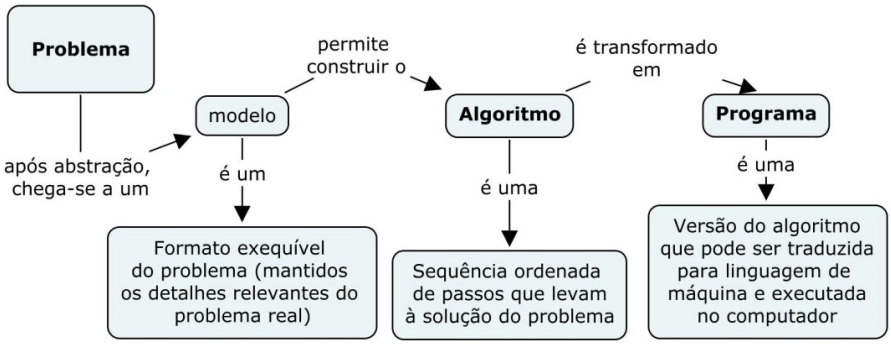
A lógica que permeia os algoritmos é introduzida a partir de uma representação gráfica usando *mapas conceituais* (do tipo *fluxograma*) e a parte prática, com testes dos algoritmos, será realizada usando a *plataforma Python*. Temos a seguir uma visão dessas ferramentas.

Mapas conceituais

Em linhas gerais, um mapa conceitual é uma representação gráfica hierárquica das relações entre conceitos, sendo estas relações indicadas por frases de ligação. Um conceito é uma noção, uma generalização, um registro rotulável, etc. Conceitos ligados por frases formam proposições e esta rede de proposições constitui o desenvolvimento de um tema sob as características inerentes a um dado conhecimento. Por exemplo, o mapa conceitual abaixo resume a história do computador com respeito ao seu modelo básico de construção (Ver Unidade I.1):



Mapas conceituais tipo fluxograma. Os mapas conceituais também servem para representar processos e esse tipo de uso terá especial utilidade aqui. Trata-se de mapas do *tipo fluxograma*[2]. Em mapas desse tipo, o rastreamento das relações entre os conceitos descreve um processo representativo da solução de um determinado problema. Em outras palavras, o mapa reflete a estrutura conceitual de uma solução. Ao se analisar mapas com essa característica, é possível perceber uma estrutura de sequência com um ponto de partida de um ponto de chegada. Tomando mais uma vez um exemplo do texto, o mapa seguinte descreve o processo de elaboração de um programa de computador (Ver Unidade I.2):



No citado exemplo, são envolvidos os conceitos de “problema”, “modelo”, “algoritmo” e “programa”, o ponto de partida é o “problema” e o de chegada é o “programa”. Também serão conceitos, substantivos que sejam convertíveis em ações simples. Por exemplo, “escrita” (substantivo), num desejado instante, representará a ação de “escrever” (verbo no infinitivo), o conceito de “leitura” deve levar à ação de “ler”, etc. Finalmente, dado o mapa conceitual acima, podemos montar uma sequência de passos que descrevem o processo de elaboração de um programa de computador da seguinte maneira: Dado um problema, elaborar um modelo,

[2] Tal como descreve este artigo:
Tavares, R. *Construindo mapas conceituais*. Revista Ciências & Cognição, Ano 04, Vol 12: 72-85. 2007. Disponível em: *http://www.cienciasecognicao.org/pdf/v12/m347187.pdf*

elaborar um algoritmo a partir do modelo e, finalmente, elaborar o programa a partir do algoritmo.

Ferramenta para elaboração de mapas conceituais. O software utilizado na construção dos mapas conceituais é o *IHMC CmapTools* desenvolvido pelo *Institute for Human and Machine Cognition da UWF-Universidade de West Florida*. Os requisitos para instalação podem ser obtidos a partir da página: http://cmap.ihmc.us/download/.

A plataforma Python e os laboratórios

Os laboratórios planejados na disciplina visam conferir os elementos sobre algoritmos e estruturas dados exibidos na parte teórica. Cada laboratório deve ser acompanhado observando-se seus objetivos, recursos e experimentação. Os objetivos descrevem o que o estudante deve atingir ao concluir os experimentos. Os recursos e experimentação fazem um paralelo entre a parte teórica e recursos disponíveis na linguagem utilizada, no caso, a linguagem Python.

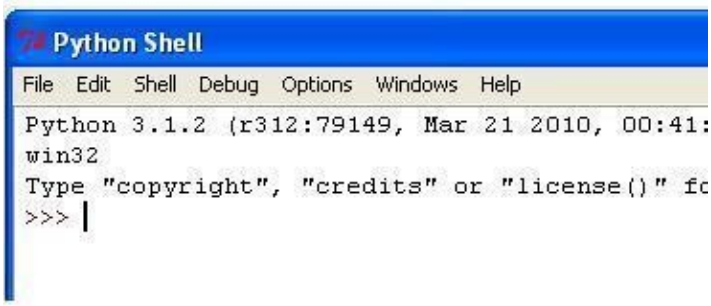
Identificação dos programas. A fim de facilitar a localização no disco, os programas são identificados com nomes iniciados com a letra L seguida de dígitos identificando a subunidade à qual estão associados e mais dois dígitos identificando o experimento realizado. Por exemplo, `L213_07.py` é o nome do programa correspondente ao Experimento 07 do laboratório da Subunidade 2.1.3.

Instalação do ambiente de programação Python. No Windows, é bastante acessar a página *http://www.python.org/download/*, baixar e instalar o arquivo com extensão "*.msi", escolhendo uma versão 3.x (tudo que é necessário é baixado automaticamente). Essa instalação vem acompanhada do aplicativo IDLE (Python GUI) de edição e execução dos programas. O sistema operacional Linux já vem normalmente com o Python instalado, mas não inclui o programa IDLE, que deve ser instalado conforme instruções especificas daquele sistema.

Após a instalação, devemos fazer um pequeno teste. Consideremos a instalação Windows. Executar o IDLE clicando em:

`Todos os programas -> Python 3.1 -> IDLE(Python GUI)`,

A execução faz surgir uma tela como a seguinte:



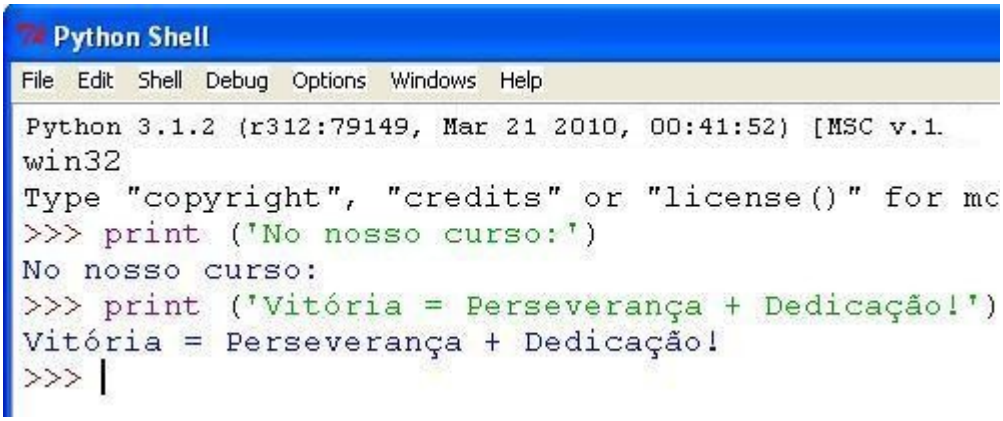
Python Shell

Essa tela, chamada de *Python Shell*, fica disponível para execução de comandos e exibição de resultados. A linguagem de programação Python é *interpretada* (Ver **Unidade 1.2**). Os comandos podem ser interpretados um a um interativamente ou reunidos em arquivos de programas chamados *módulos*.

Execução interativa de comandos. Inserimos o comando a ser executado diante do *prompt* "**>>>**". Como exemplo, vamos executar os seguintes (um de cada vez, apertando a tecla <*Enter*>):

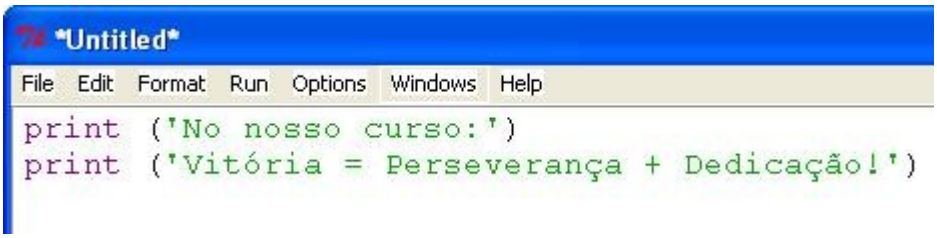
```
print ('No nosso curso:')
print ('Vitória = Perseverança + Dedicação!')
```

Cada comando executado produz seu resultado logo em seguida:



***prompt* de comandos**

Edição e execução de programas. No menu principal, clicamos em *File* (arquivo) -> *New Window* (nova Janela) e imediatamente uma tela de edição é aberta (ver figura abaixo). Nesse espaço, experimentemos com os comandos acima (já executados interativamente):



Janela de edição do IDLE

Após a digitação, *salvamos* o programa acessando o menu principal em *File* -> *Save* e escolhendo um nome, digamos, "avante.py" (escolhemos também o diretório de nossa preferência). Para executar, clicamos em *Run* -> *Run Module* (ou simplesmente apertamos a tecla F5).

Sub-Menu de execução de programas

O resultado é o seguinte:

```
>>> ================================ RE
>>>
No nosso curso:
Vitória = Perseverança + Dedicação!
>>>
```

Execução de `avante.py`

MÓDULO I

O computador e a resolução de problemas

Neste módulo abordaremos os princípios fundamentais para o uso do computador como ferramenta para a solução de problemas. Normalmente, desejando resolver um problema, precisamos apenas ativar nosso raciocínio lógico, podendo usar em seguida alguma ferramenta para levar a termo o que queremos realizar. Aqui, usaremos o computador. Por isso, devemos conhecer melhor esta máquina e, é claro, precisaremos tratar de elementos da Lógica de uma maneira um pouco mais específica.

Objetivos

- Avaliar historicamente a importância do computador e identificar sua estrutura física básica;
- Conceituar problema e identificar os princípios que permitem a resolução por computador;
- Conceituar *algoritmo* e *programa* de computador;
- Dado um problema simples, elaborar um mapa conceitual que represente o processo de resolução do mesmo, escrevendo em seguida um algoritmo a partir do mapa elaborado.

Unidades

- Unidade I.1 - O computador e seu modelo lógico
- Unidade I.2 - A resolução de problemas por computador

Unidade I.1

O computador e seu modelo lógico

Há bastante tempo a palavra "computador" deixou de ser ouvida somente em meio cientifico ou tecnológico. Está no dia a dia. Ele está presente praticamente em todos os ramos da atividade humana. Mas, a correta visão desta máquina, em primeira análise, deve ser apenas como uma sofisticada ferramenta cujo destino é auxiliar o homem na resolução dos problemas inerentes à sua existência. Constam, em particular, e historicamente, os problemas de rapidez e exatidão ao se efetuar cálculos.

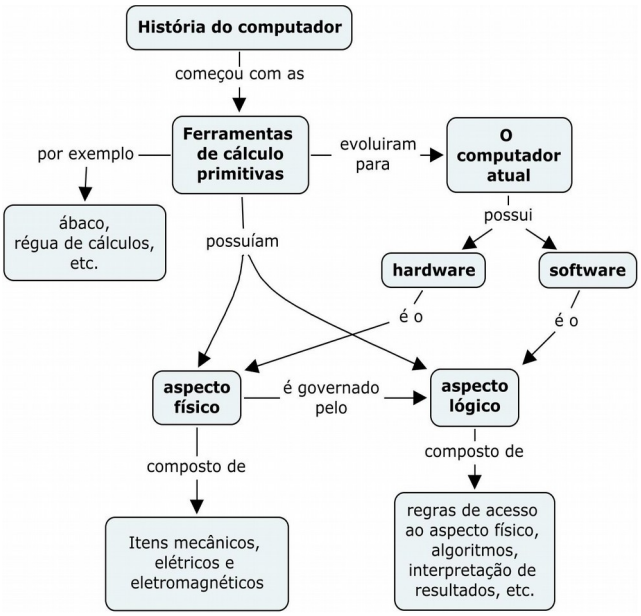
Muitas máquinas foram construídas pelo homem, mas admitimos que são precursoras dos atuais computadores apenas aquelas que permitiam automatizar procedimentos (através de cartões perfurados, por exemplo). Ao longo de sua história, o computador deixou de se fundamentar em componentes mecânicos ou puramente elétricos como também na execução de cálculos baseados em medidas físicas.

1.1.1 Histórico

A primeira máquina de uso geral e *programável* foi a máquina analítica Babbage, apesar desta ser mecânica (utilizavam-se cartões para fazê-la seguir um conjunto de instruções - *programa*). Na década de 1940, já com equipamentos eletrônicos, o primeiro passo para o computador atual foi dado por John von Neumann, ao propor que os programas fossem internos à máquina (instruções armazenadas numa *memória* – ver Bibliografia Complementar no AVA Moodle).

O computador hoje é, na verdade, um *sistema*, onde se reúnem elementos físicos, o *hardware*, e lógicos, o *software*. Atualmente, recursos de software (sistemas operacionais, aplicativos, compiladores, etc.) substituem as tarefas antigamente realizadas por hardware (época em que interagir com o hardware era corriqueiro).

Na história do computador, podemos estabelecer uma relação das ferramentas de cálculo primitivas com o computador atual, considerando que há um modelo lógico que é mantido, como mostra o seguinte mapa:



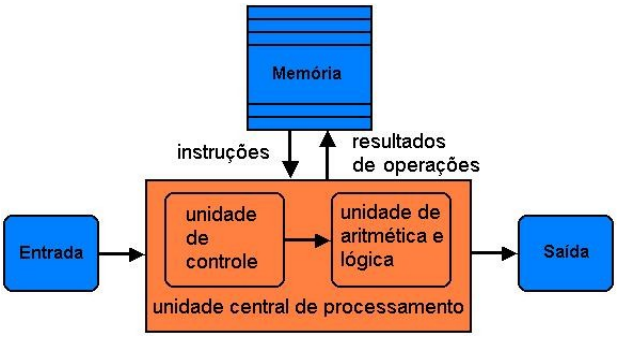
As ferramentas de cálculo primitivas e o computador atual

Fisicamente, tudo que circula internamente no computador é numericamente codificado. Tudo são *bits e bytes* (consultar a Bibliografia Complementar no AVA Moodle). Instruções e/ou dados na forma de sequências de bits, e seguindo uma coleção de regras, circulam pela máquina de uma parte a outra. As sequências de bits e as tais regras de construção compõem o que se chama de *linguagem de máquina*. Podemos dizer que esta é a linguagem nativa do computador. Assim sendo, as instruções que o computador recebe do meio externo devem ser traduzidas para esta linguagem.

Uma analogia interessante pode ser feita. Os componentes físicos do computador (o hardware) são semelhantes a operários que executam uma determinada obra. Cada um tem sua habilidade específica e só executa tarefas muito simples (no computador, seria, por exemplo, ligar-desligar, ativar-desativar, etc.). Combinando-se ações simples é possível concluir toda a obra. Um núcleo lógico, um mentor (o gerente, o projetista ou ambos), é quem detém a noção do todo e planeja conceitualmente o andamento. Na máquina, o mentor é o software produzido pelo usuário *programador*.

1.1.2 Hardware e Software

Antes de estudarmos o computador em um nível mais alto, é importante compreender ou, pelo menos, fazer uma ideia do que se passa internamente nessa máquina. O esquema de funcionamento da estrutura física do computador é normalmente chamado de *arquitetura*. Destacamos aqui a arquitetura de von Neumann, segundo a qual está baseada a maioria dos computadores atuais. Ela pode ser descrita pela seguinte figura:



Arquitetura von Neumann

Os programas que "rodam" no computador dão utilidade ao hardware e podem ser classificados como *software de base (BIOS - Basic I/O System* e o *sistema operacional)* e *software específico* (aplicativos). Para produção de resultados, a máquina segue a unidade de controle da UCP (unidade central de processamento), e esta, obedece rigorosamente um programa (software) para assumir o controle das tarefas. No momento previsto no programa, a leitura da unidade entrada (do teclado, por exemplo) pode ser acionada; a unidade de aritmética e lógica pode ser solicitada, se algum cálculo precisar ser feito; dados podem ser enviados para impressão (no monitor, por exemplo) etc.

Tradução de programas

A sequência de instruções seguida pela unidade de controle reside na memória do computador em linguagem de máquina. Os tradutores são justamente os programas que permitem a conversão de códigos escritos numa dada *linguagem de programação* para linguagem de máquina. As linguagens de programação surgiram como algo compreensível para o homem (dando-lhe melhores condições para implementação de seus projetos), porém traduzíveis para algo compreensível para o computador (já que o computador só entende linguagem de máquina!).

Linguagens de programação envolvem codificação especifica para elaborar programas. Mas, reúnem propriedades de linguagem verdadeiramente (possuem inclusive sintaxe e semântica) mesmo não sendo naturais. De acordo com proximidade da linguagem humana ou da máquina, as linguagens de programação são normalmente classificadas como de *alto nível* no primeiro caso e de *baixo nível*, no segundo.

Os programas tradutores são classificados em *montadores*, *compiladores* e *interpretadores*. Os montadores são tradutores de linguagens de baixo nível e os compiladores e os interpretadores são usados para traduzir as linguagens de alto nível. Por exemplo, a linguagem *assembly* é uma linguagem de baixo nível e é traduzida pelo montador *assembler*. Linguagens de alto nível como FORTRAN e C, são compiladas e a linguagem Python é uma linguagem interpretada.

Um interpretador analisa e executa imediatamente cada comando do programa de entrada (programa fonte). Mantém-se ativo sem gerar um arquivo diretamente executável pelo

sistema operacional (.exe). Por outro lado, um compilador analisa o código do programa fonte por completo e gera um executável (só produz o executável se não houver nenhum erro).

A memória e o significado físico de *variável*

O termo "memória" pode ser usado genericamente para indicar as partes do computador onde se armazenam dados e programas. Existe, então, a chamada *memória principal* e as demais, *secundárias*.

O acesso à memória principal é rápido e o armazenamento só persiste o tempo de um processamento da UCP (é a memória RAM - *Random Access Memory*). Há um outro tipo de memória, chamado de memória ROM (Read Only Memory). Com a evolução da tecnologia, a palavra ROM perdeu até seu significado original (de “somente leitura”). As atuais memorias *flash* são usadas para abrigar o BIOS responsável pelo boot do computador e estão também nos aparelhos eletrônicos portáteis como *smartfones*, câmeras digitais, pendrives etc.

A memória principal armazena tanto instruções quanto os dados utilizados durante a execução de um programa. Aproveitamos para introduzir aqui o conceito de *variável*. O software pode agir sobre o hardware alocando espaços de memória. Esses lugares na memória são chamados de variáveis. Uma variável tem *nome* (ou *identificador*), *tipo* (relativo ao formato e seu tamanho em bytes) e *endereço* (número associado ao lugar da variável na memória). O nome, o tipo e o endereço de cada variável ficam registrados no computador em uma *tabela de alocação*. É acessando esta tabela que o computador pode recuperar o conteúdo de uma variável recebendo apenas o seu nome.

Exercício de autoavaliação

Elabore um mapa conceitual que liga os seguintes conceitos: “linguagem de máquina”, “linguagem de programação de alto nível”, “linguagem de programação de baixo nível”, “Compilador”, “Interpretador”, ”Montador”, “programa”, “Python”. Se desejar, acrescente outros conceitos que você considere que completam o contexto. Discuta no fórum dos conteúdos da semana, comparando seu mapa com os resultados dos colegas (consulte a Bibliografia Complementar sempre que for preciso).

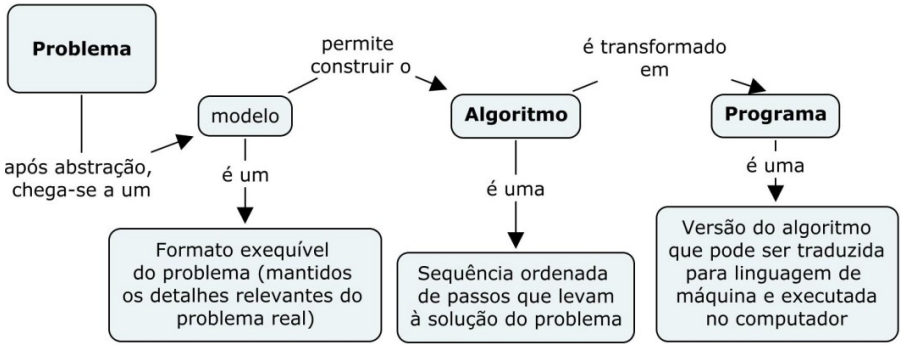
Unidade I.2

A resolução de problemas por computador

Vimos que a unidade de controle do sistema computador funciona seguindo determinadas instruções previamente especificadas no software. O conjunto destas instruções instaladas no computador (codificadas em linguagem de máquina) é, efetivamente, o *programa*. Quando executado, o programa produz os resultados correspondentes à solução de um *problema*.

Por "instrução", entende-se um comando ou uma ordem dada ao computador e está associado a uma *ação* específica. Uma ação, por sua vez, representa uma intervenção que modifica um estado inicial obtendo um claro, objetivo e previsível estado final.

Programação se inicia no confronto com o problema real, analisando o que é relevante, chegando a um *modelo* realizável. A partir do modelo, uma coerente sequência de ações consideradas inteligíveis pelo computador pode ser montada. Esta sequência é o *algoritmo*. O programa é, finalmente, a implementação (codificação) do algoritmo numa linguagem de programação. Esses elementos estão reunidos no mapa conceitual abaixo.



A programação de computadores

1.2.1 Do problema ao programa

O que, finalmente, é um problema? Talvez não tenhamos uma pronta definição, todavia é possível adiantar que, se temos algo para resolver (ou providenciar) então este "algo" é o problema. Em todas as oportunidades em que tivemos de tomar alguma decisão, encontrar respostas, produzir algum resultado, etc., nos flagramos cobrando uma saída satisfatória para todas estas questões.

Dado um problema, por onde se começa sua resolução? Na verdade, não há uma "receita" pronta para tanto. Há sugestões e algumas metodologias disponíveis, mas todas solicitam aplicação de raciocínio lógico, o que é bastante natural. Em nossa vida, desde que tomamos noção do nosso entorno, sempre tivemos que resolver problemas. Desde tomar,

quando criança, o primeiro gole d'água usando o copo (e não a mamadeira), ou amarrar os sapatos pela primeira vez, até os problemas mais “difíceis” da nossa vida adulta. Frisemos, porém, que amarrar os sapatos é sim um problema difícil diante do conhecimento infantil, e muito "fácil" no nosso universo de conhecimento.

Muitos problemas como os de Matemática, de Física, etc., foram companheiros nossos nas disciplinas regulares da vida de estudante. Outros são bem conhecidos na vida cotidiana, como localizar o número de telefone numa agenda. Em todos eles as estratégias utilizadas para a solução são as mais variadas. No caso da agenda, por exemplo, podemos encontrar o número telefônico até varrendo exaustivamente todas as informações escritas!

Resolução de problemas e os paradigmas de programação

É fato que, na busca por um modelo que represente bem a realidade e leve à solução do problema, uma estratégia deve ser adotada. Podemos chamar essas estratégias de *paradigmas*. “Paradigma”, no dicionário Aurélio, significa "modelo", "padrão". Em computação, os paradigmas são formas particulares que dão suporte às maneiras de abordagem dos problemas e de formulação das respectivas soluções.

É importante que se compreenda o conceito de paradigma nessa fase, mesmo que não seja de maneira aprofundada. A título de esclarecimento, dentre outros existentes no mundo da programação, tomemos esses dois paradigmas como exemplo (ver quadro abaixo): o paradigma *estruturado* (ou, *imperativo*) e a *orientação a objetos*.

	No paradigma estruturado	**Na orientação a objetos**
A realidade é vista como:	Um conjunto de tarefas a se cumprir	Um conjunto de *objetos*, que devem interagir entre si
Para se resolver um problema:	Deve-se montar sequências de ações que levam à solução	Deve-se construir objetos (um objeto tem dados e ações na sua definição) e fazê-los "cooperar" visando a solução
É típico deste paradigma:	Criar e preencher variáveis que representam estados (valores que mudam ao longo de uma execução) e montar blocos de instruções particulares para alterar variáveis.	Criar classes de objetos (propriedades dos objetos), construir objetos e a malha das interações entre objetos (objetos mudam de estado).
Diante de um problema a pergunta-chave é:	*Que ações se devem planejar para resolvê-lo?*	*Como os objetos em jogo se combinam* para resolvê-lo?

O paradigma que usaremos nessa disciplina será o estruturado, que, historicamente, tem sido vitorioso em termos de ensino de programação. Construir *sequências de ações que resolvem problemas* tem se mostrado coerente com a maioria das atividades que o iniciante está acostumado a lidar.

Problema computável, lógica e o conceito de algoritmo

Tomemos o exemplo já citado de localizar um número de telefone numa agenda telefônica. Podemos destacar o seguinte: uma lista de telefones é finita e, portanto, com

certeza, existe uma solução para a busca de um número nessa lista. Estamos tratando de um problema em que, quando uma sequência finita de ações for seguida e finalizada, a solução será encontrada com certeza. Nesse caso da agenda, a solução seria:

Ler cada linha da agenda, buscando-se o nome registrado na mente de quem procura
Se encontrar,
ótimo, o número do telefone desta pessoa será extraído;
Senão,
ótimo também e o resultado será "não existe este dado na agenda".

Isto é, sendo isso desejável ou não para quem busca, não há a possibilidade de uma busca infinita!

Esta classe de problemas muito nos interessa. Trata-se dos chamados problemas *computáveis*. É isso mesmo. Nem tudo pode ser transferido para a máquina e se esperar que esta encontre uma solução.

Particularmente, há um ramo da ciência, a *Lógica,* que é bastante útil na busca de soluções. O objetivo da Lógica é usar de formalidade para representar o raciocínio lógico. A questão básica é: examinando-se as leis do pensamento e formas do pensar, que ações são válidas em cada caso? Então, a lógica surge como poderosa ferramenta para a elaboração de programas. Dessa maneira, direcionamos a terminologia e assim podemos falar em uma *Lógica de Programação*, que é a aplicação da lógica na solução de problemas computáveis.

Nessa oportunidade, introduzimos um conceito fundamental em computação, que é o conceito de *algoritmo*. Por exemplo, o conjunto das palavras grifadas acima, que descreve um roteiro para busca de um número numa agenda é um algoritmo. Ou seja, a noção mais intuitiva de algoritmo é de uma “receita” para a solução do problema.

Só existe algoritmo para a solução de problemas computáveis. Ou, reciprocamente, um problema é computável se existe um algoritmo para sua solução. Conclusivamente, um algoritmo é uma sequência ordenada de instruções (também chamadas de *comandos*, ou *passos*) que, quando executada, se verifica a solução do problema. É a lógica de programação o que permeia os algoritmos.O conceito de algoritmo é amplo e é aplicável aos propósitos mais diversos.

A fase final da solução de um problema via computador é a implementação do algoritmo chegando-se ao correspondente programa. Ou seja, um programa é uma tradução da solução algorítmica para uma linguagem de programação. Podem ser usadas diversas dessas linguagens e cada uma delas possui seu conjunto particular de funcionalidades que pode ser aproveitado nessa fase.

Implementar em que linguagem?

A escolha de uma linguagem de programação depende de sua adaptação à tarefa que se quer realizar. Todas as linguagens têm suas limitações. Inclusive, é possível que

determinadas instruções sejam facilmente implementáveis numa linguagem e não sejam em outra. O conhecimento sobre linguagens e seus paradigmas pode ajudar nesta decisão.

Nessa escolha, a questão da produtividade passou a ser preponderante. As linguagens cada vez mais se afastam de codificações incompreensíveis e, de certa forma, mantêm uma tendência para o domínio da aplicação: há linguagens consideradas comerciais, de aplicações em redes de computadores etc. Surgiram também linguagens de uso geral como a linguagem C, Python, C++, Java. Estas duas últimas são orientadas a objetos. A linguagem Python, mesmo estando sobre base orientada a objetos, permite a escrita de programas claramente estruturados.

Vejamos a seguir um algoritmo bem simples que resolve o problema: “Ler um número natural **n** e mostrar o dobro desse número”. São feitas implementações desse algoritmo em PASCAL, C e Python. Mesmo não havendo conhecimento aprofundado dessas linguagens nesse momento, podemos fazer uma rápida comparação entre elas. Consideremos o algoritmo seguinte:

```
Início
    Ler o número n;
    Escrever “o dobro de n é”, 2*n;
Fim-Algoritmo
```

Onde * representa a operação de multiplicação.

Sua implementação na linguagem C:

```c
#include <stdio.h>
int main()
{
   int n;
   printf("Digite um numero inteiro: ");
   scanf("%d",&n);
   printf("O dobro de %d e %d\n", n, 2*n);
   return 0;
}
```

Sua implementação na linguagem PASCAL:

```
program dobro;
var
   n: integer;
begin
   write('Digite um número inteiro: ');
   read(n);
   writeln('O dobro de ', n, ' é ', 2*n);
end.
```

Implementação na linguagem Python:

```
n = int(input('Digite um número inteiro: '))
print('O dobro de', n, 'é', 2*n)
```

Observando os resultados acima, podemos destacar o seguinte com relação aos programas e as linguagens, entre outros aspectos: As palavras assinaladas em negrito são reservadas. Têm significado específico e devem ser escritas rigorosamente daquela forma; Nos programas, **n** é uma variável para armazenar um número inteiro; Em PASCAL e C esta variável teve que ser declarada (**int** em C, **integer** em PASCAL), já em Python isto não foi necessário (embora seja do tipo **int** automaticamente, segundo especificações desta linguagem); Os comandos para leitura e escrita também são diferenciados.

Porque Python?

Numa análise mais geral dos programas mostrados na seção anterior, em termos de tamanho e simplicidade de código, percebemos que é flagrante a vantagem da linguagem Python sobre as outras. A palavra "vantagem" está sendo usada aqui para expressar o interesse em códigos que praticamente não ofereçam empecilhos para o iniciante e os comandos possam ser verificados um a um.

A linguagem de programação Python é de uso geral. Criada por Guido van Rossum em 1990, já nasceu com afinidades na comunidade cientifica. Existem softwares que incorporam interpretadores desta linguagem, como é o caso do software de computação gráfica *Blender* (*http://www.blender.org/*), o OpenOffice (*http://www.openoffice.org.br/*), etc. A linguagem Python tem recursos suficientes para se criar qualquer tipo de software.

Além das marcantes características acima mencionadas, é também importante citar que há hoje uma considerável comunidade de usuários e que eles são muito bons colaboradores. Por tudo isso, podemos afirmar que a linguagem Python é uma excelente plataforma para principiantes.

Exercício de autoavaliação

Realize os exercícios abaixo (consulte a Bibliografia Complementar sempre que for necessário) e discuta no fórum dos conteúdos da semana. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Descubra o que é o jogo das "Torres de Hanói". Usando somente raciocínio lógico, jogue à vontade em *http://www.ime.usp.br/~leo/imatica/programas/hanoi/index.html*! Pesquise e diga se esse problema é computável (justifique sua resposta).

2 - Instale o ambiente Python no seu computador e realize as tarefas a seguir (para obter detalhes da instalação, acesse na introdução deste material o guia sobre os laboratórios e a linguagem Python). Obs.:. Este exercício tem o objetivo apenas de avaliar a compreensão sobre do que significa "implementação". Mais adiante, faremos um estudo mais detalhado sobre variáveis e comandos.

a) Abrir o ambiente IDLE e executar interativamente os comandos:

```
>>> print ('Estou de bem com Python!')
>>> print ('5 + 4 =', 5+4)
>>> print ('5 * 4 =', 5*4)
```

b) Ainda no ambiente IDLE, execute a implementação em Python do exemplo mencionado nesta subunidade (código copiado abaixo):

```
n = int(input('Digite um número inteiro: '))
print('O dobro de', n, 'é', 2*n)
```

O resultado obtido é compreensível para você, mesmo mesmo que ainda não tenhamos feito um estudo detalhado dos comandos da linguagem Python? Numa primeira análise, conseguiu descobrir o que fazem os comandos **`input()`** e **`print()`**?

1.2.2 O processo de elaboração dos algoritmos

Temos usado a palavra “receita” para dar a noção intuitiva de algoritmo. De fato, devemos seguir alguns passos até chegarmos à solução do problema. É importante notar certos detalhes na elaboração dos algoritmos. Por exemplo, nem sempre uma "receita" é realizada sem contratempos e decisões precisam ser tomadas se ocorrerem. Os "ingredientes" precisam atender aos requisitos, e as variações durante o processamento devem ser previstas.

Dependendo do problema a ser resolvido, em um algoritmo, talvez seja necessário fazer comparações, colocar condições (uso da lógica) ou fazer repetições (fazer iterações até que uma condição faça interromper). Isto corresponde às *estruturas lógicas* que integram os *algoritmos estruturados*. Estruturas ajudam a disciplinar as atitudes durante o desenvolvimento.

Estruturas lógicas

Por seus objetivos, as estruturas lógicas são *estruturas de controle* (termo que lembra a unidade de controle do computador). As estruturas de controle recebem as denominações de *estruturas de seleção* e *estruturas de repetição:*

-Utilizamos uma *estrutura de seleção* quando desejamos que determinado grupo de ações somente seja executado se determinada condição for satisfeita.

-Utilizamos *estruturas de repetição* quando iterações tiverem que ser feitas para se chegar à solução. Numa estrutura de repetição, um bloco de ações é repetido e o exame de uma condição é que define a quantidade de iterações.

Conhecer bem as estruturas lógicas é o primeiro e mais importante investimento de quem inicia o aprendizado de programação. Utilizaremos para introdução aos algoritmos o uso de *mapas conceituais* (ver introdução desse texto da disciplina). Pretende-se que as estruturas lógicas sejam “percebidas” nos exemplos a seguir. Faremos um estudo específico a partir do **Módulo III** desse curso.

O ***Exemplo 1.1*** termo objetivo de ajudar na compreensão dos elementos conceituais envolvidos na elaboração dos algoritmos. O problema a ser resolvido é, simplesmente, dar um

telefonema. Veremos que é preciso construir um modelo adequado. Em outras palavras, a realidade deve ser simplificada adequadamente.

A elaboração do algoritmo parte de um mapa conceitual do processo de execução de um telefonema. Este é convertido em seguida para uma linguagem textual. A conversão é feita explicitando-se as ações associadas às relações entre os conceitos e frases de ligação.

Exemplo 1.1

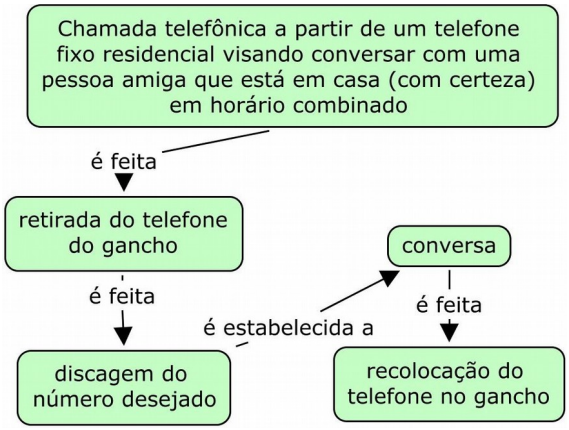
Problema: Fazer uma chamada telefônica a partir de um telefone fixo residencial visando conversar com uma pessoa amiga que está em casa (com certeza) em horário combinado (Obs.: Veremos mais adiante que o fato de ser "... uma pessoa amiga que está em casa em horário combinado" interfere razoavelmente na construção do algoritmo).

A partir do enunciado podemos extrair as características seguintes:

Dados disponíveis: O telefone e o número a ser chamado.

Saída esperada: A conversa plenamente estabelecida com a pessoa cujo número do telefone foi dado.

Os conhecimentos necessários para resolver este problema ressumem-se naqueles sobre o equipamento a ser utilizado (o telefone) e seu modo de utilização. Sabemos que a "conversa" somente acontecerá após a "discagem do número desejado" (e o contato com a pessoa desejada é estabelecido). Também, para "discagem do número desejado" é necessário que se faça a "retirada do telefone do gancho", e, a "recolocação do telefone no gancho" é feita somente após a "conversa". A solução, portanto, pode ser descrita através do mapa abaixo:



• ***Obs.****: Acesse o vídeo com a descrição desta solução no AVA Moodle, que mostra como usar mapas conceituais na elaboração de algoritmos.*

Convertendo o mapa acima em algoritmo (observemos os verbos no infinitivo), temos:

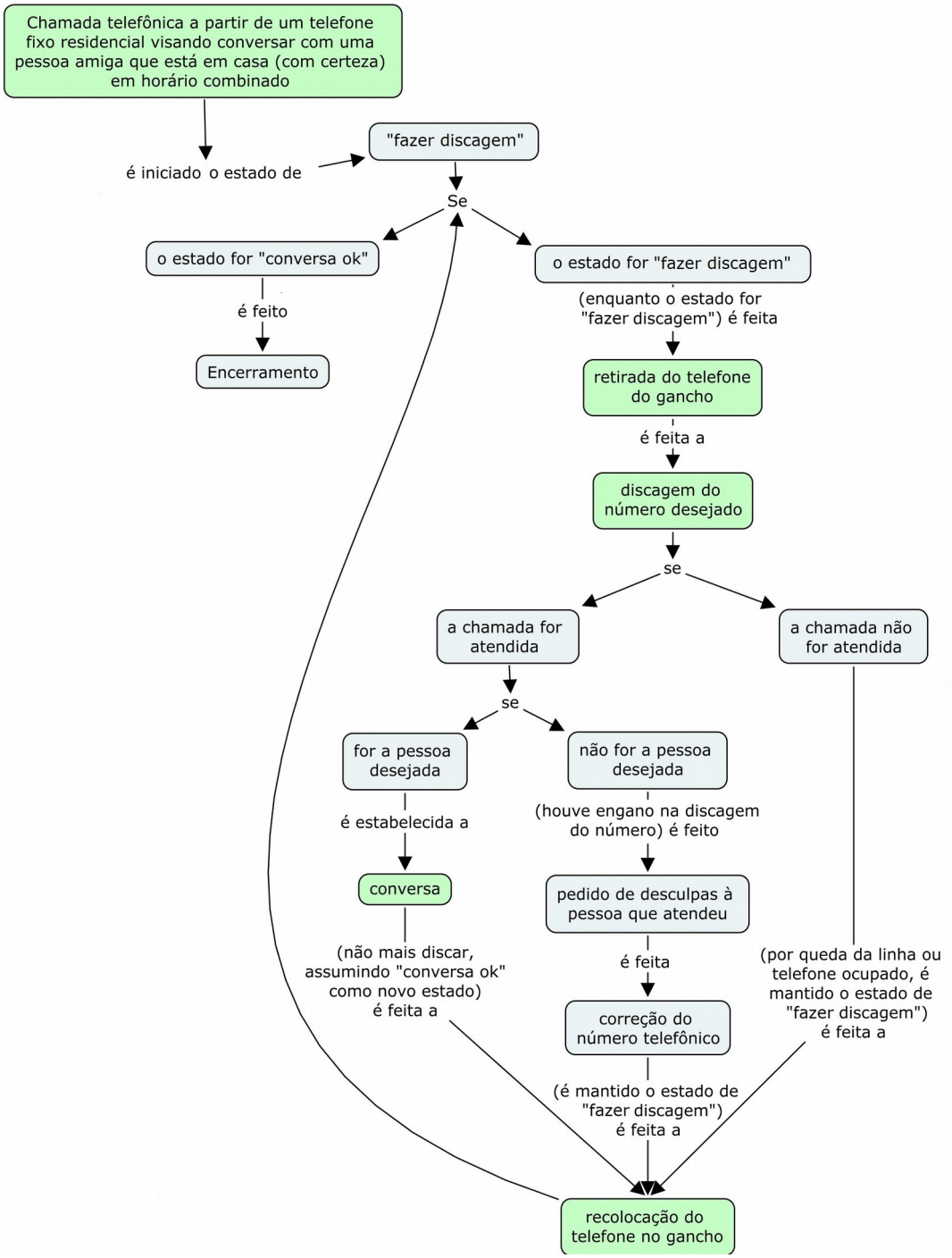
```
Algoritmo
1 Tirar o telefone do gancho;
2 Discar (ou, teclar) o número desejado;
3 Conversar (saudação, conteúdo da conversa e despedida);
4 Recolocar o telefone no gancho.
Fim-Algoritmo
```

O algoritmo em questão possui uma estrutura chamada de *estrutura linear*, ou, *sequencial* (não há "desvios"). Esta solução considera que todas condições foram atendidas sem “contratempos”. Isto é: o telefone e a linha estão funcionando bem e o número a ser discado é válido e é discado corretamente, alguém atende realmente à chamada etc. Ainda, é descartada a possibilidade de a pessoa se recusar a atender!

Bem, nem sempre tudo deve ocorrer perfeitamente. Tornando o caso um pouco mais realista, experimentemos então como ficaria a solução, admitindo-se a chamada não ser atendida ou, em sendo atendida, não ser a pessoa com quem se quer fazer o contato.

Na nova solução, proposta a seguir, um estado inicial de "fazer discagem" é adotado e este levará à "retirada do telefone do gancho" (marcando o início do telefonema). O estado "conversa ok" levará ao "Encerramento" do telefonema.

Podemos observar que, após a "discagem do número desejado", a "conversa" somente acontecerá se "a chamada for atendida" e "for a pessoa desejada". Quando a "conversa" acontecer o estado será "conversa ok" e pode ser feita a "recolocação do telefone no gancho". Por outro lado, será mantido o estado de "fazer discagem" se "a chamada não for atendida" ou se "não for a pessoa desejada". Isso estabelece uma referência circular após "recolocação do telefone no gancho" (pois esta sempre volta ao “Se”) e nova discagem será feita.



A solução algorítmica também ganha uma nova versão. Dessa vez, determinadas condições devem ser verificadas com uso de estruturas lógicas. No algoritmo abaixo:

O passo 1 define um estado inicial de “fazer discagem”;

O passo 2 é uma estrutura de repetição (representa a referência circular indicada no mapa conceitual acima). 2.1, 2.2 e 2.3 serão executados repetidamente enquanto a conversa não acontecer. O passo 2.3 é uma estrutura de seleção. Esta inclui outra também de seleção para conferir se é a pessoa desejada. O estado inicial de “`fazer discagem`” será modificado para "`conversa ok`" se a conversa se estabelecer.

```
Algoritmo
1 estado ← "fazer discagem";
2 Enquanto o estado for "fazer discagem" faça (2.1, 2.2 e 2.3):
2.1   Tirar o telefone do gancho;
2.2   Discar o número desejado;
2.3   Se a chamada for atendida então:
          Se corresponder à pessoa desejada, então:
              Conversar;
              estado ← "conversa ok" (não voltar a fazer discagem);
              Recolocar o telefone no gancho;
          Senão (se não corresponder à pessoa desejada):
              Pedir desculpas à pessoa que atendeu;
              Corrigir o número telefônico;
              Recolocar o telefone no gancho;
Senão (se a chamada não for atendida – queda da linha, telefone ocupado):
          Recolocar o telefone no gancho;
Fim-Algoritmo
```

Observação: Convém que se note um certo "alinhamento" vertical dos comandos nas estruturas, com um recuo predefinido. Isto se chama *endentação*[3]. Trata-se de um recurso usado em favor da compreensão da sequência dos comandos.

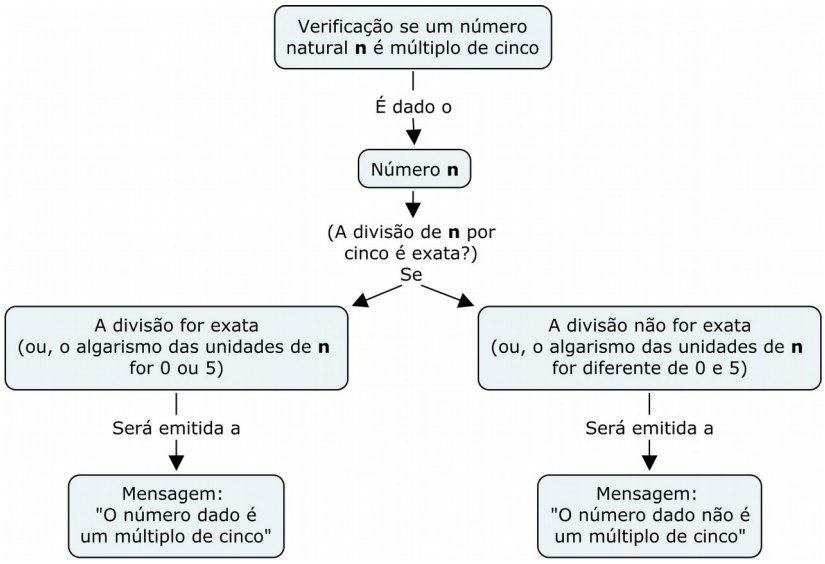
Processador humano *vs*. Computador

O ***Exemplo 1.1*** permitiu mostrar os vários aspectos de um problema cuja resolução é feita tipicamente por uma pessoa. Vamos ver agora um problema que, propositalmente, destinaremos a dois processadores diferentes: uma pessoa (no ***Exemplo 1.2***) e o computador (no ***Exemplo 1.3***). O problema é:

Dado um número natural, é este número um múltiplo de cinco?

O conhecimento necessário para a solução desse problema é que um número natural será um múltiplo de cinco se for divisível por cinco, isto é, se a divisão de tal número por cinco for exata (divisão com resto igual a zero), e isso ocorre com números cujo algarismo das unidades é zero ou cinco. Desde já, esperamos para uma pessoa e para o computador, que as características das instruções sejam diferentes, chegando a distintos algoritmos. Que capacidades são cobradas dos processadores para que o tal problema seja resolvido? Genericamente, mapa conceitual mostra como deve se encaixar qualquer solução dada para o problema:

[3] Na verdade, *endentação* é um antigo termo usado em tipografia para indicar linhas de recuo de texto. Em computação, a endentação (onde também se aceita o termo *identação*) é aplicada nos algoritmos e nos códigos dos programas.



Isto é, um número **n** é dado inicialmente. No mapa, o "Se" representa a "apuração" do resultado da pergunta: "A divisão de **n** por cinco é exata?". A partir daí, apenas um caminho será seguido: o que mostrará a mensagem "O número dado é múltiplo de cinco" ou o que mostrará "O número dado não é múltiplo de cinco".

Exemplo 1.2

Problema: Mostramos um número natural para uma pessoa e perguntamos: este número é um múltiplo de cinco?

Dado disponível: Um número natural.

Saída esperada: "É um múltiplo de cinco", ou, "Não é um múltiplo de cinco".

Observamos que o conhecimento requerido é matemático (sobre divisibilidade). Aprendemos em nossa vida escolar também a obter uma resposta imediata apenas observando o número dado, apenas observando o algarismo das unidades do número dado. Construímos então o algoritmo abaixo:

```
Algoritmo
1 Ler um número natural
2 Se o algarismo das unidades do número dado for 0 ou 5 então:
      Responder "O número é um múltiplo de cinco"
  Senão:
      Responder "O número não é um múltiplo de cinco"
Fim-Algoritmo
```

Vejamos a solução seguinte:

Exemplo 1.3

Problema: Fornecemos um número natural para o computador e solicitamos que ele informe se o tal número é um múltiplo de cinco.

Dado disponível: Um número natural.

Saída esperada: "É um múltiplo de cinco", ou, "Não é um múltiplo de cinco".

O conhecimento requerido é o mesmo do problema resolvido por uma pessoa. Mas, o que é extremamente fácil para um humano pode não ser uma tarefa simples para a máquina. Uma pessoa não precisa efetuar cálculos resolver esse problema (conforme ***Exemplo 1.2***). No entanto, para elaborarmos o algoritmo para o computador, melhor é explorar o campo das operações matemáticas. O algoritmo abaixo, então, resolve o problema:

```
Algoritmo
1 Ler um número natural
2 Se o resto da divisão do número dado por 5 for igual a 0 então:
      Escrever "O número é um múltiplo de cinco"
  Senão:
      Escrever "O número não é um múltiplo de cinco"
Fim-Algoritmo
```

Podemos comparar os algoritmos acima sob vários aspectos. Destaquemos os modos de admissão dos dados disponíveis e exibição da informação sobre a solução do problema:

	Uma pessoa	**O computador**
Admissão dos dados (Entrada)	Normalmente procura o dado de entrada (perguntando, por exemplo).	Apenas aguarda que o usuário entenda a mensagem de solicitação e insira o dado no dispositivo de entrada (por exemplo, o teclado).
Exibição do resultado (Saída)	A resposta dada por uma pessoa pode ser verbalmente, escrevendo num papel, etc.	O computador usa um dispositivo de saída para "escrever" a resposta do problema (a tela do monitor, por exemplo).

Aprendemos que a elaboração de algoritmos merece uma adequada atenção (correta sequência, legibilidade de comandos, endentação, etc.), então podemos pensar agora na "linguagem" para esta escrita. Embora a linguagem exibida nos algoritmos mostrados acima seja natural, notamos que a organização do "texto" é especial. Uma boa prática é aplicar frases ainda menores usando uma maior quantidade de símbolos.

O uso de símbolos na escrita de algoritmos é interessante porque isto reduz o texto e orienta mais diretamente a elaboração futura dos programas. Esses símbolos e abreviações são reunidos numa linguagem particular, comumente denominada de *linguagem algorítmica* (ou, *pseudocódigo*). Por exemplo, o algoritmo abaixo não perde em compreensão em relação ao algoritmo que resolve o problema dado no ***Exemplo 1.3***:

```
Algoritmo
1 Ler n
2 Se (n mod 5 = 0) então:
      Escrever n, "é um múltiplo de cinco"
  Senão:
      Escrever n, "não é um múltiplo de cinco"
Fim-Algoritmo
```

Nesse algoritmo, `n` representa `"um número natural"`. A expressão `n mod 5` significa `"resto da divisão do número dado por 5"`. O símbolo "**`mod`**" é usado nos

algoritmos como um operador para obter o resto da divisão entre dois números inteiros, como veremos mais adiante.

‘Boa’ modelagem, ‘bom’ algoritmo

Em busca de uma resolução satisfatória, o contexto de um problema deve ser dominado realmente por quem pretende resolvê-lo. Isto é, os conceitos e as relações entre estes devem fazer parte de sua estrutura cognitiva. A solução do problema repousa sobre essa estrutura, que é vista agora como etapas de um processo. O “bom” algoritmo será resultado de um processo de busca do “bom” modelo para o problema. Além do domínio do assunto, o sucesso na produção de um algoritmo é dependente da conveniente simplificação da realidade.

Utilizando casos bastante simples, os dois exemplos seguintes tratam de como a modelagem é determinante para a solução de problemas. ***Exemplo 1.4*** abaixo traz um problema que é modelado por uma única e simples operação matemática! Admitindo-se que os dados de entrada possam conter erros (pois somente valores positivos fazem sentido para o contexto do problema), a solução inclui uma “proteção” quanto a isso, só permitindo a continuação do processo se a entrada estiver correta.

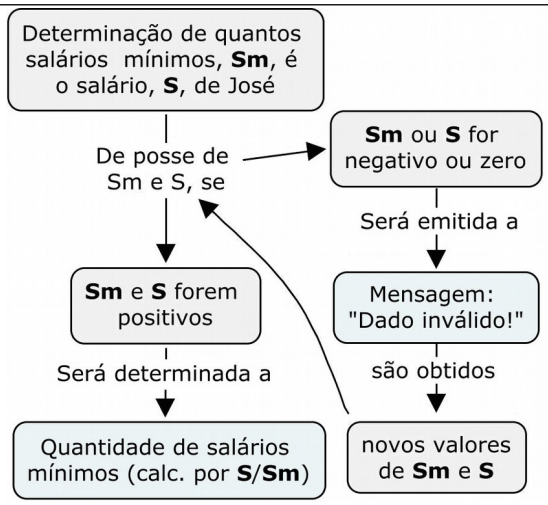
Exemplo 1.4

Digamos que o trabalhador José seja um velho conhecido nosso e desejamos saber: A quantos salários mínimos corresponde o salário de José?

Para responder esta pergunta é preciso saber antes: Qual é o salário de José? Qual é o valor atual do salário mínimo? Então, o salário de José e o valor atual do salário mínimo são as entradas para a solução do problema.

Simbolicamente, podemos representar o salário de José por **S**, o salário mínimo, por **Sm** e a resposta à questão básica (quantos salários mínimos ganha José) pode ser indicada por **S/Sm** (**S** dividido por **Sm**). Portanto, uma simples divisão entre números reais resolve o problema.

Percebemos que valores negativos ou zero para **S** ou **Sm** não fazem sentido para o contexto do problema e ainda arriscam uma divisão por zero! Vamos então admitir que possa haver erro na inserção dos dados. A solução do problema pode então ser descrita pelo seguinte mapa:



O algoritmo seguinte atende o relacionamento dos conceitos acima.

```
Algoritmo
1 Ler S
2 Ler Sm
3 Enquanto (S ≤ 0 ou Sm ≤ 0):
3.1   Escrever "Dado inválido!"
3.2   Ler S
3.3   Ler Sm
4 Escrever n, "José recebe", S/Sm, "salários mínimos"
Fim-Algoritmo
```

Notemos que há uma referência circular no mapa, que somente será interrompida *se* **Sm** e **S** forem positivos. Em outras palavras, os comandos dentro do ciclo continuam sendo executados "enquanto" o valor lido **Sm** ou o **S** for negativo ou zero. Portanto, a repetição nasce da avaliação de um "Se".

No algoritmo, o passo 3 é uma estrutura que repete a emissão da mensagem de erro `"Dado inválido!"` e novas leituras (comandos 3.1, 3.2 e 3.3, respect.) enquanto pelo menos um dos valores de entrada (**S** ou **Sm**) não for um número positivo. Quando a estrutura cessar a repetição, o algoritmo vai para o passo 4, que mostra o resultado solicitado.

Podemos conferir nas soluções dos problemas mostrados anteriormente, que usamos variáveis para gravar estados, valores preparativos para a solução final. Podemos citar:

-No ***Exemplo 1.1***, uma variável de estado foi usada como marcação de quando tentar nova ligação ou que a conversa já aconteceu;

-Nos exemplos ***1.2*** e ***1.3***, a variável **n** guardou o número inteiro como entrada no processo para que fossem feitas operações sobre aquele número;

-No ***Exemplo 1.4***, as variáveis **Sm** e **S** guardaram os valores necessários para determinar a quantidade de salários mínimos. ***Observação:*** Nesse exemplo em particular, havendo a intenção de deixar bem claro o significado da operação de divisão, uma variável (por

exemplo, **Qsm**) poderia ser criada para guardar a quantidade de salários mínimos antes de ser escrita, assim:

```
Algoritmo
1 Ler S
2 Ler Sm
3 Enquanto (S ≤ 0 ou Sm ≤ 0):
3.1   Escrever "Dado inválido!"
3.2   Ler S
3.3   Ler Sm
4 Qsm ← S/Sm
5 Escrever n, "José recebe", Qsm, "salários mínimos"
Fim-Algoritmo
```

Muitas vezes precisamos manipular até uma quantidade grande de valores. Como definir as variáveis nessas situações? Quantas variáveis seriam necessárias? Preparar variáveis é uma rotina no paradigma estruturado. Nas últimas semanas dessa disciplina, abordaremos uma das maneiras de resolver casos assim, que é reorganizando os valores em *estruturas de dados*. Por enquanto, podemos ver no exemplo a seguir, ***Exemplo 1.5***, um tipo de problema cujo modelo é possível de ser tratado nessa primeira fase do curso.

Exemplo 1.5

Conforme podemos deduzir a partir dos exemplos anteriores, se desejarmos produzir um algoritmo para somar, digamos, dois números, precisaremos pelo menos de duas variáveis para armazenar esses números (por exemplo, **a** e **b**). A solução, obviamente, será a operação de adição dessas variáveis (**a** + **b**). Como faríamos se tivéssemos uma quantidade qualquer de números, somente conhecida no momento do fornecimento dos dados?

Consideremos então o seguinte problema: Calcular a soma de uma quantidade determinada de números inseridos no computador via teclado. Primeiramente, a quantidade é informada e em seguida a leitura de dados é efetuada. A soma dos números é exibida na tela do computador após a inserção do último número.

Pensemos nas variáveis. De antemão, podemos criar uma variável para guardar a quantidade de números (a chamemos de **qtde**) e outra para a soma deles (a chamemos de **Soma**). Não temos como criar uma variável diferente para cada número lido, pois a quantidade deles somente será conhecida com o início do processo (mesmo assim, poderia exigir uma quantidade impraticável de variáveis!). No entanto, podemos criar uma (a chamemos de **n**) para representar genericamente cada número que será somado. Isto é, a variável **n** mudará de valor para cada número que entra e, consequentemente, o valor da variável **Soma** também mudará (partindo de um valor zero, irá acumulando os valores somados a cada nova leitura).

Pensemos nas ações. Por enquanto, sabemos apenas que o valor da **Soma** é zero (pois, no início, nenhum número foi somado ainda). O processo só começa de verdade com o conhecimento da quantidade de números (a variável **qtde** deve ser logo preenchida pelo

usuário do algoritmo), seguindo-se com a leitura de um dos números **n**. Assim que é lido, esse número é acumulado em **Soma**. Isto é,

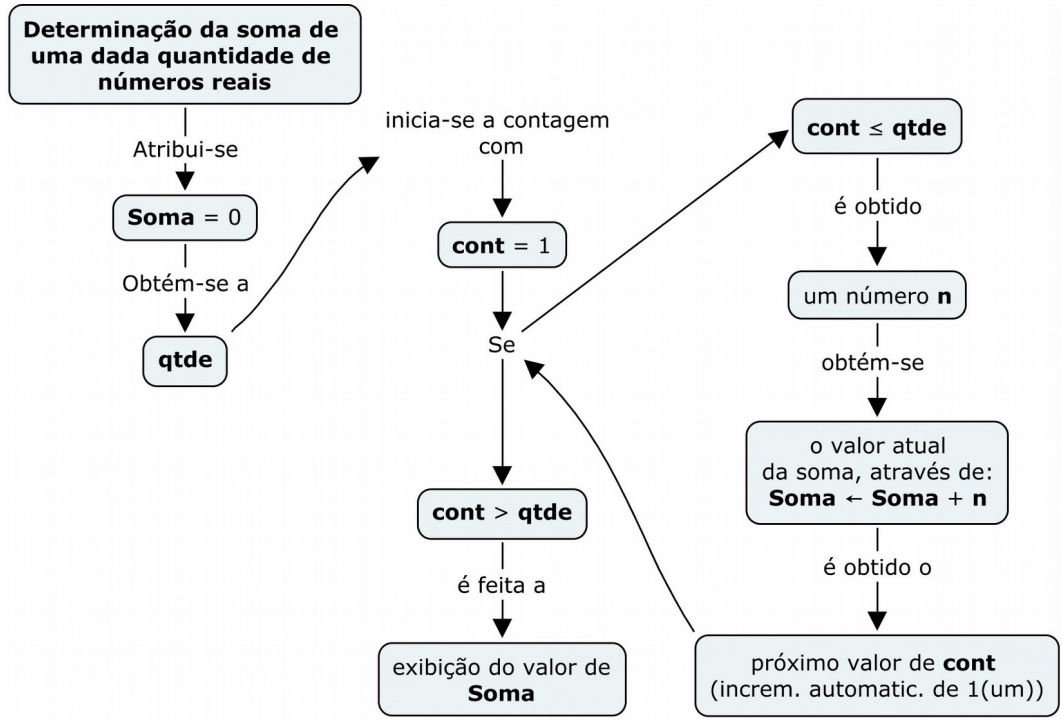
$$\mathbf{Soma}_{\mathbf{Atual}} \leftarrow \mathbf{Soma}_{\mathbf{Anterior}} + \mathbf{n},$$

o valor atual de **Soma** passa a ser seu valor antigo acrescido do valor **n** lido atualmente. Nos acostumando com esse formato (novo valor à esquerda e o antigo, à direita) podemos resumir escrevendo:

$$\mathbf{Soma} \leftarrow \mathbf{Soma} + \mathbf{n},$$

Isso vai acontecer uma, duas, três, quatro,... **qtde** vezes. Então, temos necessidade de uma variável (a chamemos de **cont** – lembrando a palavra "contador") para guardar cada valor intermediário da contagem até atingir o total previsto **qtde**. Ou seja, a cada leitura e soma de um número **n**, **cont** assumirá um valor acrescido de 1 (um) automaticamente. Ora, o algoritmo deve prever sempre nova leitura e soma de mais um número **n** se **cont** ainda for menor que **qtde**, até que cont se iguale a essa quantidade planejada.

Finalmente, o resultado armazenado na variável **Soma** estará pronto para ser exibido. Esse processo pode ser representado no mapa abaixo:



O algoritmo seguinte descreve textualmente a solução acima.

```
Algoritmo
1 Soma = 0
2 Ler qtde
3 cont = 1
4 Enquanto (cont ≤ qtde):
4.1   Ler n
4.2   Soma ← Soma + n
4.3   cont ← cont+1 (incrementar cont)
```

```
5 Escrever "A soma dos números é ", Soma
Fim-Algoritmo
```

Podemos notar uma referência circular no mapa, que somente será interrompida *se* **cont** for maior que **qtde**. Isto é, **cont** passa pelos valores 1, 2, 3, 4,... até **qtde**. Em outras palavras, os comandos dentro do ciclo continuam sendo executados "enquanto" a variável **cont** não esgotar seus valores possíveis.

Um fato interessante pode ser observado quanto ao incremento da variável **cont**, que é automático, gerando uma sequência regular de valores. Sendo assim, o algoritmo pode ser reescrito simplificando a linguagem da seguinte maneira:

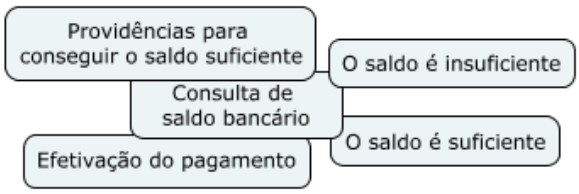
```
Algoritmo
1 Soma = 0
2 Ler qtde
3 para cont ← 1, 2, 3...qtde, faça:
3.1   Ler n
3.2   Soma ← Soma + n
4 Escrever "A soma dos números é ", Soma
Fim-Algoritmo
```

Ou seja, os passos 3 e 4 da versão anterior foram reunidos no passo 3 dessa nova versão. Esta última diz diretamente que a repetição acontece para cada um dos valores assumidos da variável **cont**.

Exercício de autoavaliação

Realize os exercícios abaixo (consulte a Bibliografia Complementar sempre que for necessário) e discuta no fórum dos conteúdos da semana. Compare seus resultados com os dos colegas participantes. Tire suas dúvidas e, oportunamente, auxilie também.

1 - O itens abaixo não elementos de um processo de "pagamento de uma despesa num banco" (um boleto bancário, por exemplo). Organize logicamente esses elementos e utilize os mesmos na elaboração de um mapa conceitual do processo. Construa o correspondente algoritmo da solução.



2 - No Edital de convocação de matrícula de uma universidade consta o seguinte "roteiro para matrícula":

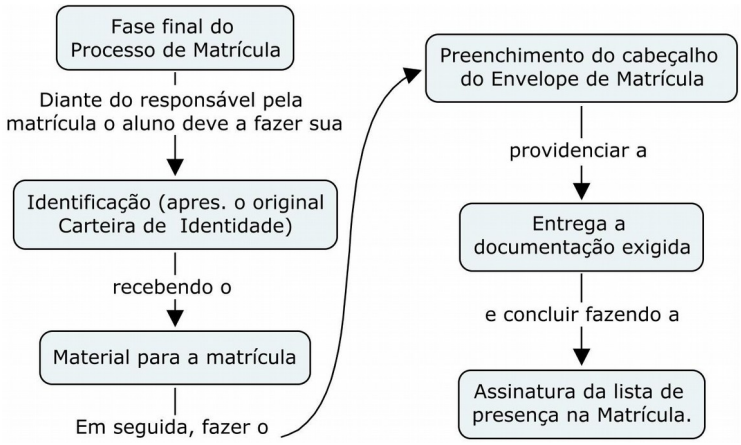
"1. Identificar-se diante do responsável pela matrícula, apresentando o ORIGINAL DA CARTEIRA DE IDENTIDADE e receber o material para a matrícula;

2. Preencher o cabeçalho do Envelope de Matrícula;

3. Entregar a documentação exigida ao responsável pela matrícula;

4. Assinar a lista de presença na Matrícula."

Observamos que esse roteiro se refere, na verdade, ao momento final (ou, "fase final" do processo de matrícula), quando o aluno já está com sua documentação pronta e se apresenta ao responsável pela matrícula. Aplicando a ideia proposta nesse texto, o citado roteiro pode ser dado pelo seguinte mapa:



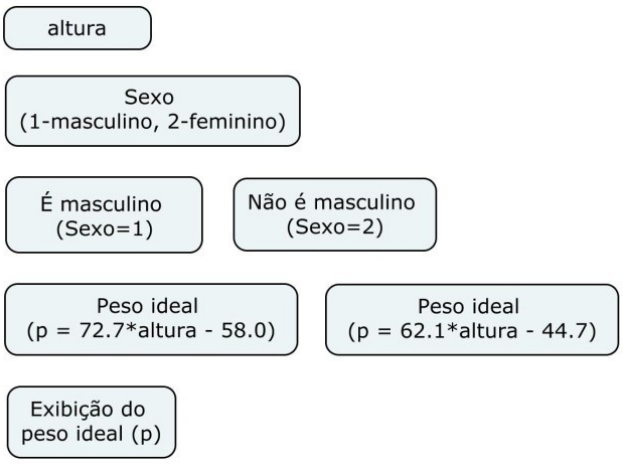
Sabe-se que, de fato, há uma "fase inicial" onde o aluno deve fazer: "Preenchimento da ficha de cadastro via Internet", "Conferência da documentação (inclusive a ficha de cadastro impressa)". Se algum documento estiver faltando, deve ser providenciado.

Sem modificar o mapa da fase final, acrescentando os elementos da fase inicial, faça uma versão completa do mapa do processo de matrícula.

3 - Pergunta-se a uma pessoa quanto tempo ela gasta numa viagem de sua cidade para a capital de seu estado dirigindo seu próprio carro. Bem, se essa pessoa já viveu essa experiência, ela terá seguramente uma resposta, pois tem noção da distância percorrida e a velocidade aproximada que costuma viajar. Tomemos então o seguinte problema geral que seria resolvido por computador: Determinar o tempo de viagem entre duas cidades (em horas), dada a distância (em km) entre elas, e uma estimativa da velocidade média em km/h.

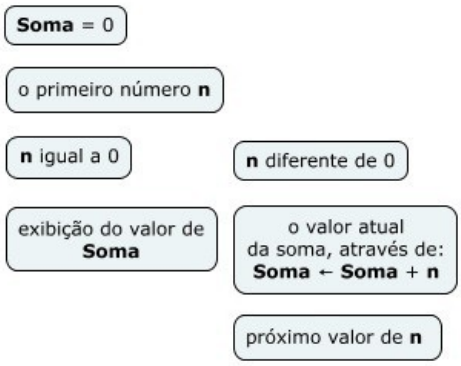
A proposta de modelagem desse problema deve vir dos conhecimentos básicos de Física. Sabe-se que o tempo gasto, **t**, é calculado por **t = d/v**, onde **d** é o deslocamento e **v** é velocidade média. Para simplificar a solução vamos admitir somente valores positivos nesse processo, mas os dados serão digitados por uma pessoa que pode se enganar. "Inspire-se" no ***Exemplo 1.4*** e escreva um mapa da solução desse problema e o algoritmo correspondente.

4 - Seja o seguinte problema: Dada a altura (em metros) de uma pessoa e seu sexo (1-masculino ou 2-feminino), calcular e escrever seu peso ideal `p` (em kg), utilizando as fórmulas empíricas: `p = 72.7*altura - 58.0`, se for homem e `p = 62.1*altura - 44.7`, se for mulher. Elabore um mapa conceitual que inclua os sugestivos elementos abaixo, e o algoritmo correspondente da solução:



Observação: Os dados serão fornecidos para o computador via teclado e não serão inseridos valores inválidos (como altura negativa ou zero, ou, então, dados sobre sexo diferentede masculino ou feminino).

5 - Consideremos o seguinte problema: Calcular a soma de uma quantidade *indeterminada* de números (diferentes de 0-zero) inseridos no computador via teclado. A soma dos números é exibida na tela do computador após a inserção do último número, o zero, que não entra nos cálculos (é usado apenas para marcar o fim da leitura).



Podemos observar que a solução deste problema pode ser inspirada naquela do ***Exemplo 1.5***. Podemos usar um conjunto semelhante de variáveis e temos pequenas e interessantes variações: Aqui, por exemplo, a quantidade de números não é conhecida nem precisamos dela para calcular a soma! O primeiro número **n** deve ser lido inicialmente (já que seu valor definirá a continuação do processo ou não). O processo de soma e leitura do próximo número **n** continua *enquanto* o número digitado for diferente de zero. Aproveite a "dica" abaixo para elaborar o mapa da solução e o correspondente algoritmo.

MÓDULO II

Elementos básicos para elaboração dos algoritmos

A lógica aplicada na construção dos algoritmos é efetivada mediante a manipulação de dados, variáveis (espaços para armazenamento na memória do computador), expressões etc. As expressões são combinações de dados, variáveis e *operações.* Os dados e as variáveis são os *operandos,* de modo semelhante ao que conhecemos em Matemática. Conforme forem os tipos dos operandos envolvidos, as expressões podem ser *aritméticas*, *lógicas* ou *literais*. Este módulo trata dos tipos de dados, do significado das variáveis, dos comandos para acesso à memória do computador visando armazenamento ou exibição de conteúdos, e do uso de expressões para representar as soluções de determinados problemas.

Objetivos

- Identificar o dado e seus tipos como matéria prima para a informação;
- Definir variável e sua relação com o hardware;
- Identificar os comandos elementares dos algoritmos;
- Combinar dados, variáveis e seus tipos, para a construção de expressões;
- Representar soluções de problemas através de expressões aritméticas, lógicas e literais.

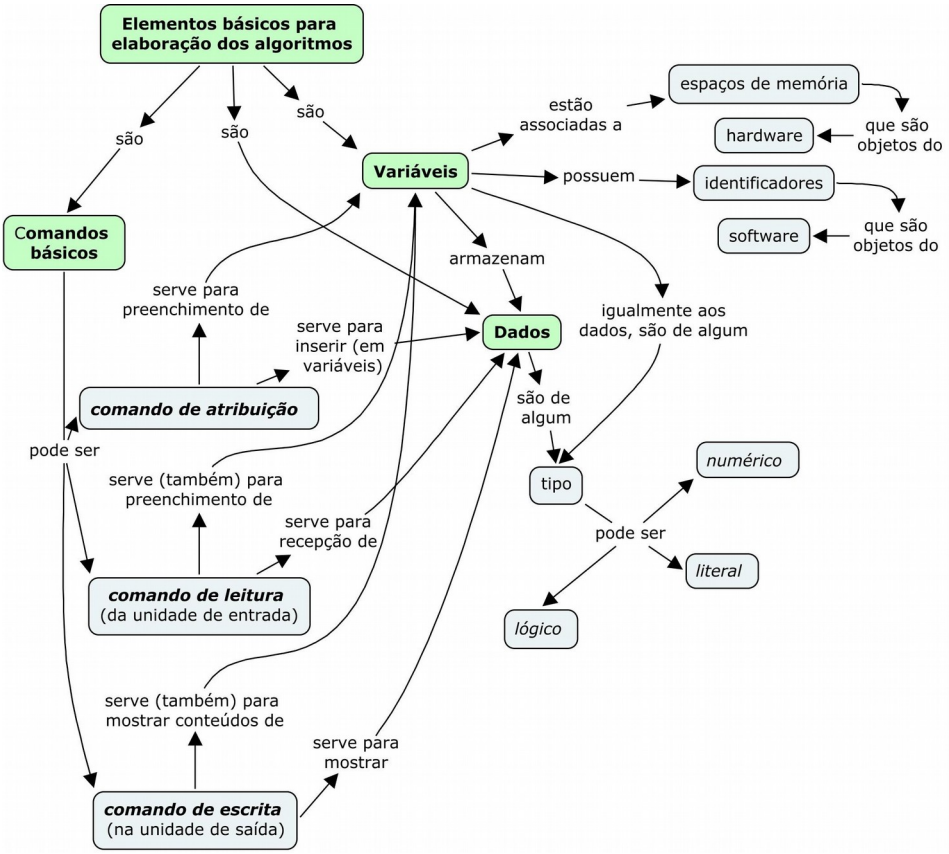
Unidades

- Unidade II.1 - Dados, variáveis, comandos básicos
- Unidade II.2 - Expressões

Unidade II.1

Dados, variáveis, comandos básicos

Nesta unidade relacionaremos itens considerados indispensáveis para a elaboração dos algoritmos. São os elementos envolvidos quando usamos o computador como ferramenta para resolução de problemas. Foi feita uma escolha de determinadas regras (menos formais e mais baseadas na experiência do autor) e conceitos que, com certeza, serão presenças bastante comuns. Ou seja, o conteúdo dessa unidade inclui o que, de algum modo, será aplicado nos algoritmos. São conhecimentos sobre tipos de *dados*, *variáveis* e os considerados *comandos básicos* que são: o comando de atribuição e os comandos para leitura e escrita de dados. Teremos a seguir um detalhamento desses conceitos (inclusive com a realização de experimentos em laboratório) que, de antemão, podem ser interligados conforme o seguinte mapa conceitual:



2.1.1 Tipos de dados

Objetivamente, o dado é a matéria prima para o computador cumprir suas finalidades. São exemplos: uma medida de um certo volume, o faturamento mensal de uma empresa, uma sequência temporal de valores de medidas pluviométricas, a relação dos nomes dos alunos aprovados numa disciplina etc. Por conta de suas origens o computador sugere uma tendência

para números. Contudo, cada vez mais, novos tipos de dados passam a ser manipulados por esta máquina (basta considerar os formatos para manipulação de som e imagem, por exemplo).

O fato é que, para ser processado, o dado precisa de uma representação. Já dizemos anteriormente que um tipo de dado tem a ver com a quantidade de bytes necessários para representá-lo. Porém, é mais que isso. Não é só a quantidade de bytes que diferencia dois tipos de dados, mas a organização dos mesmos também, que obedece a um critério e cada sistema providencia essa estrutura. Sem se aprofundar por enquanto na representação binária interna, é possível classificar os tipos básicos manipuláveis nos algoritmos. É claro que nos programas haverá uma maior especialização por causa da utilização de recursos de cada linguagem. Nos algoritmos, podemos resumir os tipos básicos em: Numérico, Literal e Lógico.

O tipo numérico

Quando falamos em tipo numérico estamos nos referindo basicamente aos números inteiros e aos números reais, porque são incorporados à maioria das linguagens de programação. Por exemplo, se formos tratar da idade das pessoas, costumamos fazê-lo usando números inteiros (15 anos, 23 anos, 60 anos etc.) e quando tomamos medidas da altura fazemos isto usando valores que normalmente não são inteiros (pois consideramos frações de um metro), como 1,70 m, 1,76 m etc. Tomando esse último caso como exemplo, decidimos então que todas as alturas devam ser medidas com números reais (se uma pessoa tiver uma altura de 2,0 m, por exemplo, é só escrever com parte decimal igual a 0,0).

Os números inteiros correspondem a elementos do bem conhecido conjunto **Z** {...-3, -2, -1, 0, 1, 2, 3, ...} da Matemática e os números reais a elementos do conjunto **R**. Apenas para lembrar, o conjunto dos números reais é o resultado da união dos racionais (que podem ser escritos na forma de fração) e dos irracionais (que não podem ser escritos na forma de fração). Sabemos também que os inteiros são racionais (porque podem ser associados a frações com denominador igual a um) e por isso estão também dentro dos reais.

Devemos observar que **Z** e **R** são conjuntos infinitos, mas suas representações no computador são finitas. Ou seja, existem números inteiros e reais que simplesmente não existem no computador! Então, como se resolve essa questão? Quando se tenta calcular um valor e este não está entre as opções representáveis pela máquina, o computador simplesmente assume um dos valores representáveis aproximado como resposta.

Uma decorrência da finitude das representações pelo computador, é que este só consegue trabalhar com números decimais exatos. Os números irracionais (os decimais não exatos nem periódicos - como o número PI, *3,1415...*) e os racionais que resultam em decimais periódicos (como 1/3 que equivale à dízima periódica *0,333...*) são aproximados para decimais exatos (as últimas casas decimais são abandonadas). Na prática, o que se pode fazer para melhorar a situação é combinar recursos de hardware e software para aumentar ao máximo a quantidade de números representáveis. Consegue-se dessa maneira um "menu" mais rico,

minimizando os erros (já que não se pode evitá-los!). Ou seja, quanto mais números forem representáveis no computador menos erros se cometem.

Exemplo 2.1

Os números seguintes são exemplos de inteiros: 15, -8, 546, +12, -235, 0, 3180, 2147483647.

Os números reais são indicados no computador de modo que apareça um ponto decimal (e não vírgula como escrevemos corriqueiramente). Os números reais ainda podem ser escritos numa notação em potência de dez.

Exemplo 2.2

Os números seguintes são exemplos de números reais: 5627.45, 234., -14.5279, 3.14, -5.4, 542., 27.3, 5.627238e+5, 7.1e-2.

5.627238e+5 significa 5.627238 x 10^5 e é a escrita de 562723.8. O número 7.1e-2 significa 7.1x 10^{-2} é a representação em potência de dez do número 0.071.

A codificação binária (interna) é diferente para inteiros e reais por isso, nos programas, inteiros e reais correspondem a tipos distintos.

O tipo literal

Muitas vezes precisamos manipular ou armazenar caracteres que podem ser dígitos, letras e outros símbolos. Usamos então o tipo literal. Os caracteres podem estar isolados ou formando *cadeias de caracteres*. Este é o tipo usado para representar textos (a partir do mais simples – com apenas um caractere). O que importa é que a sequência literal tenha algum sentido para o problema. Todo valor literal deve estar colocado entre aspas. Assim, quando forem escritos dessa maneira, até mesmo numerais serão tratados como valores literais.

Exemplo 2.3

Os valores abaixo são do tipo literal: “Alagoas“, “12 + 4 = 16”, “a”, “ ” (branco), “Oi! Tudo bem?”.

O tipo lógico

Quando formulamos uma sentença (por exemplo, "A Terra tem apenas um satélite natural", "5 é um número ímpar" ou "4 + 3 > 8" etc.) estamos afirmando coisas que podem ser verdadeiras ou falsas. Estamos falando das *proposições* (conjunto de palavras ou símbolos expressando um sentido completo), que pertencem ao estudo da Lógica. O resultado do

julgamento (verdadeiro ou falso) corresponde ao chamado *tipo lógico*. Dizer que uma sentença é verdadeira (respectivamente, falsa) equivale a dizer que seu valor lógico é *verdadeiro* (respectivamente, *falso*). Isto é, o tipo lógico tem apenas dois valores possiveis.

Laboratório - Tipos de dados

Objetivos

Fixar os conceitos relativos aos tipos de dados básicos (numéricos, lógicos e literais).

Ampliar estes conceitos a partir da identificação dos tipos implementados na linguagem Python.

Recursos e experimentação

Na linguagem Python os tipos de dados são divididos em *primitivos* e *definidos pelo usuário*. Os definidos pelo usuário, combinações dos primitivos com o objetivo de atender a uma aplicação em particular, não serão estudados nessa disciplina. Os tipos primitivos já nasceram pertencendo ao núcleo da linguagem e podem ser *simples* e *compostos*. Trataremos mais adiante dos tipos compostos (como as listas e os conjuntos). Os simples (numéricos, literais e lógicos) merecem a primeira e especial atenção. Para facilitar o estudo, iremos conferir dos tipos no interpretador de comandos da linguagem, usando interativamente a função predefinida `type()`. É só digitar diante o sinal "**>>>**" (chamado de *prompt*) a função `type()`, inserindo o valor desejado entre os parênteses e a tecla `ENTER` logo em seguida.

Os tipos numéricos. Em Python os números inteiros estão implementados como **`int`** e os números reais como **`float`**. Além destes, a linguagem Python possui uma representação dos *números complexos* através do tipo **`complex`**. Os complexos são números com parte imaginária definida por **j**, tal que $\mathbf{j^2 = -1}$. Isto é, são números do t*ipo a* + bj, onde a e b são números reais, *s*endo *a* o coeficiente da parte real e b, o coeficiente da parte imaginária.

Experimento 01

Usar a função `type()` para conferir os tipos aos quais pertencem os números: 15, 0, 0., 9147483647, 3000.0, 3e9, 527.45, -5.6e+5, *3.1* + *2j* (3.1, parte real, e 2.0, parte imaginária).

```
>>> type(15)
<class 'int'>
>>> type(0)
<class 'int'>
>>> type(0.)
<class 'float'>
>>> type(9147483647)
<class 'int'>
>>> type(3000.0)
<class 'float'>
```

```
>>> type(3e9)
<class 'float'>
>>> type(527.45)
<class 'float'>
>>> type(-5.6e+5)
<class 'float'>
>>> type(3.1 + 2j)
<class 'complex'>
>>>
```

Observação: Notamos que o resultado produzido pela a função `type()` se inicia com a palavra "class", que pode parecer incompreensível nesse momento. No entanto, podemos adiantar que tal palavra está associada ao paradigma fundamental de construção da linguagem. Na verdade, os tipos predefinidos em Python são estruturas mais sofisticadas chamadas de "classes" (assunto da próxima disciplina, Algoritmo e Estrutura de Dados II). O objetivo de seu uso aqui visa apenas a conferência em laboratório das afirmações sobre tipos de dados. A versatilidade da linguagem permite que os conceitos do paradigma imperativo continuem válidos.

O tipo literal. A linguagem Python permite o tratamento de literais através do tipo **`string`** (ou, **`str`**). Os valores do tipo string são colocados entre aspas simples ou duplas. Se forem colocados entre aspas duplas, as aspas simples podem ser usadas como caracteres comuns dentro da cadeia. Reciprocamente, se a cadeia for fechada com aspas simples, as duplas serão tratadas como caracteres comuns. Quanto à codificação de caracteres, a linguagem adota, por padrão, a UTF8 (acesse Bibliografia Complementar no AVA Moodle).

Há uma farta quantidade de recursos para manipulação de `string`s, mas vamos estudá-los de forma parcelada. A linguagem Python também possui representação para caracteres não imprimíveis como: pula linha (`'\n'`), tabulação (`'\t'`) etc. Para os propósitos atuais, estas informações são suficientes.

Experimento 02

Usar a função `type()` para conferir os tipos dos dados seguintes: "Alagoas", "12 + 4 = 16", " ", "'default' = valor padrão" , 'Oi! Tudo bem?', "A", 'A'.

```
>>> type("Alagoas")
<class 'str'>
>>> type("12 + 4 = 16")
<class 'str'>
>>> type(" ")
<class 'str'>
>>> type("'default' = valor padrão")
<class 'str'>
```

```
>>> type('Oi! Tudo bem?')
<class 'str'>
>>> type("A")
<class 'str'>
>>> type('A')
<class 'str'>
```

O tipo lógico. Este tipo tem somente dois valores possíveis, conforme comentamos acima. Em Python, o tipo é definido como **`bool`** e os valores lógicos são **`True`** (verdadeiro) e **`False`** (falso). O termo bool deriva de *Boolean* (Booleano), de George Boole, matemático inglês que deu nome à álgebra que manipula dados desse tipo (a *álgebra booleana*). Mais adiante veremos um pouco desta álgebra que é bastante útil em computação. Apenas para conferir, consideremos o experimento seguinte.

Experimento 03

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

`True` (verdadeiro) e **`False`** (falso) são valores de fato e devem ser escritos exatamente dessa maneira. Isto é, iniciando com letra maiúscula e sem nenhum acréscimo de outros símbolos, pois NÃO são cadeias de caracteres simplesmente. Se forem acrescentadas aspas acontecerá o seguinte:

```
>>> type("True")
<class 'str'>
>>> type("False")
<class 'str'>
```

Exercício de autoavaliação

Realize os exercícios abaixo com base nos conhecimentos construídos nesta subunidade. Discuta no fórum dos conteúdos da semana. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Para cada frase abaixo, informe um adequado tipo de dado da linguagem Python para representar os valores sublinhados: a) Na minha casa tem uma escada com 30 degraus; b) Marque que com um "x" a resposta certa; c) A passagem de ônibus custa R$ 1,70; d) No equipamento de uma fábrica consta o aviso: "cuidado"; e) 3 + 5 = 9 (Falso) .

2 - Que acontece se executarmos o comando **`type(false)`**? Explique.

3 - Determine os tipos dos dados abaixo (entre os numéricos, lógicos e literais da linguagem Python): 59, ‘65’, “’123’”, 17.4, 8.1e3, 3.14, 9-9j, 999999999, 9999.9, “true”, ‘Oi!’ (experimente também outros valores).

2.1.2 As variáveis e a atribuição de valores

O conceito de variável, do ponto de vista do hardware, foi apresentado quando falamos no modelo lógico do computador na **Subunidade 1.1.2**. Nos informamos de que uma variável ganha nome e endereço e o programa tem o poder de criar e acessar espaços na memória usando esse conceito. Isso quer dizer que uma variável é uma abstração necessária para que o programa acesse a memória fisicamente.

Identificadores

Pelas razões que foram expostas, o nome de uma variável é sua identidade (só lembrando, o programa localiza uma variável na memória através do seu nome, consultando a tabela de alocação). Por esse motivo, o nome da variável é seu identificador, termo também usado para nomear outros elementos usados nos programas como veremos mais adiante.

A maioria das linguagens de programação tem basicamente o mesmo conjunto de regras para construção de identificadores. Adotaremos nesse curso as seguintes:

Os nomes são escritos com caracteres da codificação padrão. Se for ASCII, por exemplo, não pode haver caracteres acentuados, cedilha etc.

Os nomes deverão começar com letra ou sublinhado ("_") e a partir daí poderão conter quaisquer combinações envolvendo letras, números e sublinhado.

É terminantemente proibido deixar espaços em branco no nome da variável.

Adotaremos também a diferenciação entre maiúsculas e minúsculas. Por exemplo, os nomes aux, Aux, aUx, auX, AUX, Aux, são distintos e devem identificar variáveis diferentes.

Exemplo 2.4

São nomes distintos de variáveis: `Var`, `var`, `Nome`, `nome`, `x1`, `X1`, `X_1`, `Valor`, `Nome_do_Aluno`, `Salário` (Os seguintes são ***nomes inválidos***: 1x, *Valor, sua-media, Oi!, Nome do Aluno).

Atribuição

Quando um programa está em execução, os lugares da memória associados às variáveis podem ser preenchidos por valores diferentes. O termo "variável" tem a ver com isso também, mas há igualmente uma relação com o termo usado em Matemática. Sob determinados aspectos, porém, as variáveis tratadas nos algoritmos e programas são apenas semelhantes às variáveis matemáticas. Para explicar isso devemos falar do preenchimento de uma variável no computador que chamaremos de *atribuição*. Tomaremos as atribuições como comandos. Uma atribuição é uma maneira de preencher uma variável diretamente dentro de um programa. O exemplo seguinte resume este significado. Em Matemática, dadas duas variáveis, x e y, se tivermos a igualdade x = y, estamos afirmando que x é igual a y e, ao mesmo tempo, y é igual a x. No computador, isto pode significar uma comparação entre x e y

(verificando se x é igual a y) ou esta é apenas uma das duas possibilidades de atribuições para se conseguir que o valor de x seja igual a y.

Consideremos as variáveis x e y com algum valor inicial. Então, as duas atribuições possíveis são as seguintes. Em um programa, x = y equivale ao comando "guarde em x uma cópia do que estiver em y" (e o conteúdo de x antigo será substituído pelo de y). No outro sentido, se y = x, temos o comando "guarde em y uma cópia do que estiver em x" (e o conteúdo de y antigo será substituído pelo de x). Não devemos esquecer que, nos dois sentidos, como se trata de cópia de conteúdos, a variável que cedeu seu conteúdo permanece inalterada.

Nos algoritmos, quando queremos fazer uma atribuição indicamos pelo símbolo ← para não deixar dúvidas quanto ao significado. Portanto, aplicando esta notação, escrevemos x = y como x ← y (dizemos: x recebe o conteúdo de y) e y = x, como y ← x (dizemos: y recebe o conteúdo de x).

As variáveis e os tipos de dados

Por conta do preenchimento de variáveis via atribuição direta, é bastante conveniente que uma variável seja tratada como um depósito temporário de dados. Para facilitar a compreensão, podemos comparar, grosso modo, as variáveis com depósitos de mantimentos. Nesse sentido cada depósito tem seu destino certo. Uma vez definido o destino do vasilhame, o conteúdo tem que ser compatível. Consideremos, por exemplo, um cesto de palha, uma sacola plástica e uma garrafa de vidro que reservamos para armazenar, respectivamente, laranjas, farinha e leite. Vamos ter sérios problemas se, por acaso, colocarmos o leite no cesto ou tentarmos colocar laranjas inteiras na garrafa de vidro.

A metáfora acima tem sua utilidade. Primeiramente, serve para explicar que toda variável é reservada para um determinado tipo de dado. Reciprocamente, um dado só pode ser armazenado em variáveis que abriguem as propriedades do seu tipo. Logo, considerando os tipos estudados acima, elas podem ser, genericamente: numéricas, literais e lógicas. Em segundo lugar, a metáfora dá uma noção intuitiva de como se procede a troca de conteúdos de variáveis (ver o exemplo seguinte).

Exemplo 2.5

Vamos supor que queiramos trocar os conteúdos das variáveis numéricas x e y cujos conteúdos são x ←35 e y ← 48 atualmente. Usando apenas o comando x ← y, o objetivo não será atingido porque teremos no final x e y iguais a 48 e usando somente o comando y ← x, teremos x e y iguais a 35. A solução, portanto, é tomar uma variável auxiliar (como um terceiro vasilhame). Ou seja, antes da atribuição x ← y precisamos "salvar" o valor anterior de x na variável auxiliar (vamos chamar esta variável de aux). A sequência de comandos para a troca de conteúdo passa a ser:

aux ← x (salva o conteúdo de x);

x ← y (x recebe y);

y ← aux (y recebe o valor antigo de x, salvo em aux).

Conversões de tipos

Muitas vezes, determinados tipos de dados precisam ser atribuídos a variáveis de tipos diferentes ou os tipos de um conjunto de valores dentro de uma mesma operação precisam ser homogeneizados. Os dados então podem ser transformados no tipo desejado através de uma conversão. Internamente há uma reorganização dos bytes para tanto.

Exemplo 2.6

A atribuição seguinte preenche a variável x do tipo literal:

```
x ← "235"
```

Se usarmos um conversor para o tipo numérico, na atribuição seguinte, y é numérica:

```
y ← numérico(x)
```

Atribuição múltipla

A metáfora dos vasilhames explicou bem a troca de conteúdos de memória, mas não pode ajudar aqui porque precisamos do significado real de atribuição, que é a cópia de conteúdos.

Muitas vezes precisamos atribuir um mesmo valor a mais de uma variável. Por exemplo, se tivermos que atribuir o valor 0 (zero) às variáveis a, b e c, a maneira mais comum de fazê-lo é a seguinte:

```
a ← 0
b ← 0
c ← 0
```

Algumas linguagens de programação, no entanto, permitem que essas atribuições sejam feitas na forma de apenas um comando, da seguinte maneira:

```
a ← b ← c ← 0
```

Isto é equivalente à sequência de comandos:

```
c ← 0
b ← c
a ← b
```

Ou seja, o `0` é copiado em `c`, em seguida o `c` é copiado em `b` e depois `b` em `a`. Ao final, cada variável estará com o conteúdo igual a zero.

Laboratório - Variáveis e atribuição de valores

Objetivos

Fixar os conceitos de variáveis e de atribuição de valores às mesmas.

Estabelecer a relação entre os tipos de dados e tipos das variáveis.

Aplicar os conceitos acima utilizando os recursos implementados na linguagem Python.

Recursos e experimentação

Uma variável em Python passa a existir quando definimos um valor para a mesma. Ou seja, sua criação é automática. Esta é uma vantagem considerável em relação a outras linguagens porque não precisamos ficar preocupados em "declarar" todas as variáveis no início de cada programa. Se isto fosse necessário, qualquer esquecimento seria acusado como erro.

Identificadores. Embora a linguagem adotada nesse curso (a linguagem Python, em sua mais recente versão) admita caracteres acentuados, tentaremos ao máximo evitá-los apenas para reduzir a multiplicidade de formas e consequente risco de confusão para o iniciante. Seguiremos as regras já citadas acima nesta subunidade.

Acrescenta-se a restrição de que um identificador não pode ser *palavra reservada* (palavras da linguagem de propósitos específicos). Toda linguagem de programação possui um conjunto de palavras reservadas. No caso de Python, são seguintes:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Atribuição. A atribuição de valores às variáveis em Python é feita usando o símbolo "=" (convém reforçar que não se trata da igualdade matemática) e aceita atribuição múltipla. O experimento seguinte consiste em conferir os conteúdos de variáveis após algumas atribuições. Para fazer uma atribuição interativamente no interpretador é bastante escrever o nome da variável, o símbolo de atribuição, o valor a ser atribuído e, por último, a tecla `ENTER`. Para conferir o valor atribuído digita-se o nome da variável e a tecla `ENTER`.

Experimento 01

Criar as variáveis **`Nome`**, **`ok`**, **`a`** e **`b`**, conferindo seus valores iniciais.

```
>>> Nome = 'Maria'
>>> ok = True
>>> a = b = 0
```

```
>>> Nome
'Maria'
>>> ok
True
>>> a
0
>>> b
0
```

Como mostra o experimento acima, as atribuições de valores a variáveis distintas ocupam linhas de código também distintas. Isto se deve ao fato de que a linguagem Python usa o final da linha para indicar a conclusão do comando. A linguagem não adota explicitamente um delimitador como é feito em outras linguagens, que concluem os comandos usando geralmente um ponto-e-vírgula.

Com a atribuição múltipla, atribuímos valores a mais de uma variável ocupando apenas uma linha de código, mas isto se aplica somente quando o interesse for atribuir valores idênticos. Para atribuir valores distintos a variáveis distintas ocupando uma mesma linha do programa, a linguagem Python exibe a particularidade da *atribuição em lista*. Por exemplo, podemos preencher as variáveis **Nome**, **ok**, **a** e **b** usando um único comando:

```
Nome, ok, a, b = 'Maria', True, 0, 0
```

O experimento seguinte repete o anterior exceto pelo uso de atribuição em lista.

Experimento 02

Criar as variáveis `Nome`, `ok`, `a` e `b`, atribuindo os valores `'Maria'`, `True`, `0` e `0`, respectivamente, e conferindo em seguida os conteúdos das mesmas.

```
>>> Nome, ok, a, b = 'Maria', True, 0, 0
>>> Nome
'Maria'
>>> ok
True
>>> a
0
>>> b
0
```

Observação: A atribuição acima citada aplica, na verdade, uma propriedade do tipo predefinido *`tupla`* (ver bibliografia do curso sobre a linguagem Python). Tuplas são estruturas avançadas e não temos agora a necessidade de abordá-las integralmente. Por enquanto, é útil saber que **`(Nome, ok, a, b)`** e **`('Maria', True, 0, 0)`** são tuplas e a atribuição em lista obedece à correspondência entre a lista das variáveis e a lista dos seus respectivos valores.

As variáveis e os tipos de dados. Como a criação de uma variável através de uma atribuição é automática, é igualmente automática a definição do seu tipo. O tipo do dado atribuído define o tipo da variável. A terminologia para isso é *tipagem dinâmica*. Ou seja, o tipo da variável é o mesmo de sua última atribuição.

Tudo que foi dito a respeito das variáveis e dos tipos de dados continua valendo para qualquer linguagem de programação, porque variáveis e dados têm que ser compatíveis. Mas, tudo isso fica resolvido quando aproveitamos a "bondade" da linguagem Python, relativamente à tipagem dinâmica. Na mesma seção do experimento anterior, faremos o experimento seguinte.

Experimento 03

Conferir os tipos das variáveis criadas no experimento anterior e em seguida tentar atribuir um valor do tipo literal à variável `ok` (cujo valor inicial é do tipo lógico).

```
>>> type(Nome)
<class 'str'>
>>> type(ok)
<class 'bool'>
>>> type(a)
<class 'int'>
>>> type(b)
<class 'int'>
>>> ok = 'Tudo bem!'
>>> type(ok)
<class 'str'>
```

A variável **`Nome`** é do tipo string, **`ok`** é booleana e as variáveis **`a`** e **`b`** são inteiras. Quando foi atribuído à variável `ok` o valor literal "`Tudo bem!`", não ocorreu nenhum erro! Confirmamos então nesse experimento que a linguagem implementa uma adaptação da variável ao tipo do dado atribuído. Uma variável pode até receber mais de um tipo de dado. Nesse caso, ela assume o tipo do último valor atribuído.

Conversões de tipos. Na linguagem Python, o recurso da conversão de tipos pode ser aplicado rapidamente usando a sintaxe:

`tipo``(argumento a ser convertido),`

onde *`tipo`* pode ser, basicamente: `int`, `float`, `str` e `bool`. Quando a conversão acontece entre os tipos numéricos, lógicos ou de qualquer um para string, não há restrições. Porém, mais alguns detalhes entram em jogo quando a conversão é de um tipo qualquer para **`bool`** (lógico) ou de **`string`** para outro tipo. Valem, então, as seguintes observações:

- Na conversão de um valor numérico para **`bool`**, o zero será mapeado para **`False`** e os valores diferentes de zero para **`True`**.

- Na conversão de uma **`string`** para **`bool`** a string nula (`''`) corresponderá a **`False`** e as demais serão mapeadas para **`True`**.

- Na conversão de um valor **`bool`** para **`int`** (ou, **`float`**), o valor lógico **`True`** é mapeado para valor `1` (ou, `1.0`) e o valor lógico **`False`** resulta no valor `0` (ou, `0.0`).

- Para converter um valor literal em um número, a cadeia de caracteres deve ser compatível com o tipo a converter. Ou seja, deverá ser formada por caracteres numéricos e com ponto, se for **`float`**, e sem ponto se for **`int`**.

Experimento 04

Dada a variável literal `x`, criar a variável `y` com o valor de `x` convertido para `float`, e a variável z com o valor de y convertido para inteiro. Em seguida, conferir os tipos e os conteúdos de `x`, `y` e `z`.

```
>>> x = '235.75'
>>> y = float(x)
>>> z = int(y)
>>> type (x)
<class 'str'>
>>> type (y)
<class 'float'>
>>> type (z)
<class 'int'>
>>> x
'235.75'
>>> y
235.75
>>> z
235
```

É importante observar:

- O conteúdo de x é, de fato, diferente do conteúdo de y, pois x é uma string (é uma cadeia de caracteres) e y é um número.

- Na conversão para inteiro, a parte decimal do número simplesmente deixa de existir. Isto aconteceu com a variável z que assumiu o valor truncado de y.

Exercício de autoavaliação

Realize os exercícios abaixo com base nos conhecimentos construídos nesta subunidade. Discuta no fórum dos conteúdos da semana. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Da lista de identificadores de variáveis abaixo (que são aceitas na linguagem Python), quais deles são ***inválidos***?

a) `B_53`; b) `X_`; c) `_X`; d) `X5`; e) `5X`; f) `Notas`; g) `A_B`; h) `A&B`; i) `No._Apto`

2 - O que acontecerá se tentarmos usar uma palavra reservada como nome de variável? Use o interpretador para comprovar sua resposta.

3 - Se x, y e z armazenam os valores 5.0, -34.2 e 18.3 respectivamente, determine seus valores atuais após a seguinte sequência de comandos: x ← y, y ← z, z ← x. Use o interpretador Python para conferir o resultado.

4 - Realizando o ***Experimento 04*** acima tentando executar o comando **`y = int(x)`**, um erro ocorrerá. Dê uma explicação para este erro e confira isto usando o interpretador.

5 - Se executarmos seguidamente os comandos (criando as variáveis `ok` e `cadeia`) `ok = True`, `cadeia = ''` (aspas sem espaço entre elas), `bool(ok)`, `bool(cadeia)`, `int(ok)`, `int(cadeia)` e `int(bool(cadeia))`, o que encontraremos? Explique.

2.1.3 *Entrada e saída de dados*

Vimos que o modelo de computador que estamos abordando obedece à arquitetura von Neumann (**Unidade I.1**) e, portanto, os algoritmos que desenvolveremos têm destino certo para esta máquina. Isto é, pelo que estudamos, podemos concluir que nossa meta de encontrar a solução algorítmica de um problema obedece ao esquema:

onde o processamento é um conjunto de comandos que usa a entrada como matéria prima para produzir uma saída.

Logo, os comandos de acesso à entrada e à saída do computador devem fazer parte do processamento. Executar um comando de entrada significa "ler" valores da unidade de entrada e armazená-los em variáveis. Executar um comando de saída significa "escrever" valores constantes e/ou conteúdos de variáveis na unidade de saída. Vamos considerar aqui que o dispositivo de entrada (o que vai fornecer os dados para os programas) é o teclado e o dispositivo de saída será o monitor.

Comandos de leitura e escrita de dados

Nos algoritmos, podemos representar um *comando de leitura* (ou, *comando de entrada*) da seguinte maneira:

`ler` *identificador*

onde *identificador* é o nome da variável que receberá o conteúdo digitado.

A escrita de dados no monitor funciona como uma exibição apenas. A melhor maneira de vê-lo é como uma página que exibirá os resultados. Nos algoritmos podemos representar um *comando de escrita* (ou, *comando de saída*) de dados da seguinte maneira:

`escrever` *argumentos*

onde *argumentos* são nomes de variáveis cujos conteúdos serão copiados na tela ou, ainda, expressões ou valores constantes cujos valores serão visualizados.

Exemplo 2.7

Consideremos um processamento com apenas dois comandos. O algoritmo seguinte apenas lê um caractere e, em seguida, o exibe no monitor:

```
Algoritmo
1 ler c
2 escrever "Você digitou", c
Fim-Algoritmo
```

Este algoritmo espera que o usuário digite um caractere qualquer para mostrá-lo logo em seguida juntamente com uma mensagem. Esta espera do programa precisa combinar com a informação ao usuário de que esta espera realmente existe! Ou seja, é melhor avisar ao usuário do programa que um valor precisa ser digitado. Logo, a versão seguinte é mais completa:

Exemplo 2.8

Este algoritmo emite uma mensagem solicitando do usuário o valor de entrada.

```
Algoritmo
1 escrever "Digite um caractere: "
2 ler c
3 escrever "Você digitou", c
Fim-Algoritmo
```

Assim a comunicação com o usuário fica mais clara. A espera pela digitação continua existindo, mas a frase "`Digite um caractere`" escrita na tela não deixa dúvidas sobre o que o usuário tem que fazer nesse momento: enquanto o usuário não digitar o que foi solicitado, a execução ficará paralisada!

É o programador quem providencia as condições para o diálogo do programa com o usuário. A importância disto é que o usuário "conversa" com o programa observando a tela e não tem a obrigação de conhecer os meandros do que foi programado. Uma má comunicação pode até inviabilizar um programa (não se deve esperar que o usuário adivinhe o que o programa quer!). Assim, as mensagens e a correta coleta dos dados digitados são fundamentais. Vejamos a seguir um outro exemplo.

Exemplo 2.9

O algoritmo abaixo irá processar internamente um valor literal e um número inteiro. Mas, durante a comunicação com o usuário, estes valores ganham respectivamente os significados de *nome* e *idade* de uma pessoa (se não fosse assim as solicitações seriam "Digite um valor literal" e "Digite um inteiro"!).

```
Algoritmo
1 escrever "Digite seu nome: "
2 ler nome
3 escrever "Digite sua idade: "
4 ler idade
5 escrever nome, "tem", idade, "anos!"
Fim-Algoritmo
```

Um aspecto que também merece atenção é que muitas vezes desejamos reduzir a quantidade de variáveis com a intenção de simplificar ou de exercer maior controle sobre a possibilidade de erros. Vejamos agora um caso numérico e aproveitemos para abordar esse assunto. O algoritmo mostrado abaixo calcula e escreve a soma e a diferença entre dois números quaisquer.

Exemplo 2.10

O algoritmo pede dois números, a e b, e em seguida calcula e escreve o resultado de (a + b) e de (a + b).

```
Algoritmo
1 escrever "Digite dois números:"
2 escrever "a = "
3 ler a
4 escrever "b = "
5 ler b
6 soma ← a + b
7 dif ← a - b
8 escrever "a + b = ", soma
9 escrever "a - b = ", dif
Fim-Algoritmo
```

Observemos que os comandos 6 e 7 são atribuições de valores às variáveis `soma` e `dif` (elas recebem, respectivamente, a soma e a diferença entre os números a e b) e os comandos 8 e 9 se encarregam de mostrar seus valores na tela (acompanhados de um texto explicativo). Nesse caso, `soma` e `dif` podem ser eliminadas se resolvermos mostrar diretamente os resultados da soma e da diferença, como no ***Exemplo 2.11*** abaixo.

Exemplo 2.11

Esta é uma reescrita do algoritmo do exemplo anterior eliminando-se as variáveis `soma` e `dif`.

```
Algoritmo
1 escrever "Digite dois números:"
2 escrever "a = "
```

```
3 ler a
4 escrever "b = "
5 ler b
6 escrever "a + b = ", a + b
7 escrever "a - b = ", a - b
Fim-Algoritmo
```

Nesse caso, ao receber as instruções para escrever os resultados das expressões a + b e a - b, o computador providencia o cálculo das mesmas primeiramente visitando os lugares da memória chamados de `a` e de `b`. O resultado de cada uma é reservado temporariamente em outro lugar da mesma memória. Logo em seguida, o computador lê este lugar temporário e mostra seu conteúdo na tela.

Laboratório - Entrada e saída de dados

Objetivos

Fixar o significado dos comandos de entrada e saída, como meios para admissão de dados e exibição de resultados no computador.

Usar os comandos de entrada e saída para implementar comunicação entre o programa e o usuário.

Aplicar os conceitos acima utilizando os recursos implementados na linguagem Python.

Recursos e experimentação

Já citamos o mecanismo de leitura do computador via teclado e o significado da escrita de dados no monitor de vídeo. No computador, leitura e escrita somente acontecem quando o programa solicita. Esta solicitação é feita através de comandos de entrada e saída de dados.

Comandos de leitura. Para a entrada de dados (implementação do comando `ler`), a linguagem Python possui a função predefinida `input()`, que, primitivamente, captura valores literais da entrada padrão. Então, no programa, podemos fazer a conversão para o tipo desejado. A função já vem pronta para a comunicação com o usuário através de mensagens. Estas mensagens (dados do tipo `string`) são colocadas entre os parênteses.

Comando de escrita. Para a saída de dados (implementação do comando `escrever`), está disponível a função predefinida `print()`. Esta função possui uma interessante coleção de parâmetros que serão aqui explorados em momentos adequados. Experimentaremos inicialmente o formato mais simples. Isto é, apenas colocando entre os parênteses o que será impresso na tela.

Para realizar os experimentos seguintes, armazenaremos os programas em arquivos com a extensão **`.py`** e executaremos os mesmos usando o aplicativo IDLE.

Experimento 01

Implementar o algoritmo dado no ***Exemplo 2.7***. Criar o arquivo `L213_01.py` e digitar as seguintes linhas de

```
c = input()                 #comando 1 do algoritmo
print ('Você digitou', c)   #comando 2 do algoritmo
```

Observação: O símbolo '#' é usado para indicar um comentário. O interpretador ignora como comando o que for escrito a partir dessa marca até o final da linha.

Execução de `L213_01.py` . Executando o programa e digitando, por exemplo, o caractere '@' como dado de entrada, visualizamos:

```
>>>
@
Você digitou @
>>>
```

Aproveitando que a função `input()` aceita uma mensagem comunicação com o usuário, vamos implementar o ***Exemplo 2.8*** que é a versão melhorada do exemplo anterior. Observemos que os recursos da linguagem irão permitir a redução dos comandos 1 e 2 do algoritmo para apenas um.

Experimento 02

Implementar o algoritmo dado no ***Exemplo 2.8***. Criar o arquivo L213_02.py e digitar as seguintes linhas de c

```
c = input('Digite um caractere: ') #comandos 1 e 2 do algoritmo
print ('Você digitou', c)          #comando  3 do algoritmo
```

Execução de `L213_02.py` . Executando o programa e digitando, por exemplo, o caractere '$' como dado de entrada, visualizamos:

```
>>>
Digite um caractere: $
Você digitou $
>>>
```

Experimento 03

Implementar o algoritmo dado no ***Exemplo 2.9***. Criar o arquivo L213_03.py e digitar as seguintes linhas de c

```
nome = input('Digite seu nome: ')
idade = int(input('Digite sua idade: '))
print (nome, 'tem', idade, 'anos!')
```

Execução de `L213_03.py`. Executando o programa e digitando, por exemplo, "Joao da Silva" como o nome e 49 como a idade, visualizamos:

```
>>>
Digite seu nome: João da Silva
Digite sua idade: 49
João da Silva tem 49 anos!
>>>
```

Experimento 04

Implementar o algoritmo dado no ***Exemplo 2.11***. Nesse exemplo, o algoritmo solicita a leitura dois números sem informar se são inteiros ou reais. Aqui, vamos admitir que sejam reais (do tipo float, portanto). Criar o `arquivo L213_04.py` e digitar as seguintes linhas de

```
print ('Digite dois números:')
a = float(input('a = '))
b = float(input('b = '))
print ('a + b = ', a + b)
print ('a - b = ', a - b)
```

Execução de `L213_04.py`. Executando o programa e digitando, por exemplo, a = 40 e b = 15 como valores de a e b respectivamente, visualizamos:

```
>>>
Digite dois números:
a = 40
b = 15
a + b =  55.0
a - b =  25.0
>>>
```

`(a + b)` e `(a - b)` são expressões com números do tipo `float`. Na verdade, existem outros recursos a serem considerados quando o assunto é expressão, porém isto será abordado com mais detalhes na próxima subunidade.

Podemos explorar um pouco mais a função `print()` utilizando, por exemplo, o recurso de produzir uma saída formatada. Eventualmente, usaremos este formato quando desejarmos organizar melhor da saída do programa adaptando aos significados assumidos pelos valores em questão. Por exemplo, podemos usá-lo, entre outras aplicações, quando o resultado envolver moeda e quisermos que os números reais apresentem exatamente duas casas decimais, que são os centavos. Genericamente, o formato é o seguinte:

`print ('`*expressão_de_controle***`' % `***lista_de_argumentos***`)`**

onde a *expressão_de_controle* é uma `string` formada por caracteres a serem escritos na tela e códigos de formatação indicando como os argumentos devem ser escritos. Os argumentos são valores a serem impressos na tela e, havendo mais de um, devem estar entre parênteses e separados por vírgula. Na formatação é usado o caractere '`%`' seguido de uma das letras: `c`,

d, e, f, g, o, s, u ou x. Por exemplo: %c é o código usado para formatação de caracteres simples, %d para inteiros na base decimal, %f para números do tipo float, %e para notação científica etc.

Para conhecer melhor este formato, o experimento seguinte é realizado. São apresentados os casos de formatação de acordo com o tipo desejado.

Experimento 05

Ler um número x inteiro e escrever a saída deste número com os formatos de: caractere (%c), inteiro decimal (%d), notação científica (%e), float (%f), formato geral (%g), octal (%o) (base 8 – Ver Bibliografia Complementar), string (%s), representação interna sem sinal (%u) e hexadecimal (%x) (base 16 – Ver Bibliografia Complementar).

Criar o arquivo L213_05.py e digitar as seguintes linhas de código:

```
x = int(input('Digite um inteiro x = '))
print ("x como inteiro:......... %d " % x)
print ("x em notação científica: %e " % x)
print ("x como float:........... %f " % x)
print ("x no formato geral:..... %g " % x)
print ("x em octal 8:........... %o " % x)
print ("x como string:.......... '%s' " % x)
print ("x repr. sem sinal:...... %u " % x)
print ("x em hexadecimal:....... %x " % x )
```

Execução de L213_05.py. Executando o programa e fornecendo x = 65, por exemplo, visualizamos:

```
>>>
Digite um inteiro x = 65
x como inteiro:......... 65
x em notação científica: 6.500000e+01
x como float:........... 65.000000
x no formato geral:..... 65
x em octal 8:........... 101
x como string:.......... '65'
x repr. sem sinal:...... 65
x em hexadecimal:....... 41
>>>
```

Podemos fazer o comentário seguinte. Como inteiro, x saiu igual ao próprio valor (65). Em notação científica, o resultado se refere à igualdade $65 = 6.5x10^1$. Como float o valor de x ganhou casas decimais. E assim por diante, bastando observar que as conversões são automáticas (se a conversão não for possível o

Estes formatos ganham ainda uma sub-formatação associada à ocupação do número na tela. Embora não seja interesse nosso aprofundar este assunto agora, é interessante

considerar o caso do tipo `float`. Vimos no experimento acima que a escrita do número `65` como `float` ganhou 6 casas decimais. Este é o formato padrão. Mas, se for interesse do programador, tanto a quantidade de caracteres da apresentação quanto a quantidade casas decimais podem ser controladas. A representação `%f` ganha então dois números separados por um ponto. O primeiro, indicando a quantidade total de caracteres da apresentação final do número e o segundo, a quantidade deseja de casas decimais. Por exemplo, se indicarmos a saída de `x` com **`%5.2f`**, isto significa que o número terá cinco caracteres sendo dois reservados para duas casas decimais e um reservado para o ponto, da seguinte maneira:

6 5 . 0 0

Se este mesmo número for escrito com o formato **`%7.2f`** (com mais caracteres que o necessário) a saída será:

6 5 . 0 0

Ou seja, as casas que sobram simplesmente não são preenchidas. Por outro lado, se formatarmos com uma quantidade menor, o número resultante também não será prejudicado. Por exemplo, a formatação **`%4.2f`**, indica que devem ser reservados dois dígitos para as casas decimais mais um para o ponto. Ora, se a quantidade total dos dígitos é quatro resta apenas um dígito para representar a parte inteira! Mas o sistema adota como regra a preservação da parte inteira do número:

6 5 . 0 0

Por isso, quando não indicamos o total de caracteres da saída (se escrevermos apenas **`%.2f`**), o sistema adotará automaticamente o resultado mais adequado desde que não haja perdas na parte inteira do número. Nesse caso, a saída será a mesma acima. O experimento seguinte tem o objetivo de conferir as afirmações acima.

Experimento 06

Usando o interpretador interativamente, criar a variável numérica `x` com o valor 65 e reescrevê-la no formato `float`:

```
>>> x = 65
>>> print ("%5.2f" % x)
65.00
>>> print ("%7.2f" % x)
  65.00
>>> print ("%4.2f" % x)
65.00
>>> print ("%.2f" % x)
65.00
>>> print ("%.1f" % x)
65.0
```

O experimento seguinte mostra a solução de um problema que envolve formatação da saída.

Experimento 07

O programa abaixo lê dois números, **a** e **b**, calcula a média aritmética, **m**, e produz uma saída no seguinte formato:

A media entre xx.xxx e xx.xxx é xx.xxx

(Ou seja, "A média entre a e b é m", sendo todos os números escritos com três casas decimais). Sabendo-se que existem na linguagem os operadores '+' (adição) e '/' (divisão), como veremos na próxima unidade, a média é calculada por:

m = (a + b)/2

Criar o arquivo L213_07.py e digitar as seguintes linhas de código:

```
print ('Digite dois números:')
a = float(input('a = '))
b = float(input('b = '))
m = (a + b)/2
print ('A media entre %.3f e %.3f é %.3f' % (a, b, m))
```

Observamos que os números a serem escritos, (a, b, m), estão entre parênteses e separados por vírgula, obedecendo à ordem lógica de escrita.

Execução de L213_07.py. Executando o programa e fornecendo a = 34.32 e b = 40.215, encontramos:

```
>>>
Digite dois números:
a = 34.32
b = 40.215
A media entre 34.320 e 40.215 é 37.267
```

Exercício de autoavaliação

Realize os exercícios abaixo com base nos conhecimentos construídos nesta subunidade. Discuta no fórum dos conteúdos da semana. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Use o interpretador Python para mostrar que a atribuição **x = y = z = 2.0** cria três variáveis numéricas do tipo float com o mesmo valor 2.0.

2 - Dadas as atribuições

X = 5.913.520.000 (valor em km) e

Y = 1,32 x 10^{22} (valor em kg),

construa uma frase que envolva X, Y e as strings abaixo

'A distância média de Plutão ao Sol é de',

'e sua massa é de', e use a função `print()` para escrevê-la. Experimente usando a função `print()` para escrevê-la (use o interpretador Python interativamente).

3 - Considere a tarefa de ler duas variáveis numéricas, a e b, escrever seus valores recém lidos, em seguida, trocar seus conteúdos e reescrevê-las, mostrando seus novos valores. Elabore duas versões de programas que cumpram essa mesma tarefa, uma para cada critério a baixo:

a) Fazer a troca de conteúdo das variáveis conforme foi apresentado no ***Exemplo 2.5*** (ou seja, usando uma variável temporária auxiliar);

b) Fazer a troca de conteúdo entre as variáveis aplicando a atribuição em lista (vista no ***Experimento 02*** da **Subunidade 2.1.2**), sabendo-se que o uso deste recurso permite que a troca dos conteúdos seja feita da seguinte maneira. Dadas duas variáveis **`x`** e **`y`**, seus conteúdos podem ser trocados através da atribuição (simples assim!):

```
x, y = y, x
```

(Obs.: A armazenagem temporária fica por conta do mecanismo da própria linguagem Python).

4 - Os salários dos trabalhadores em certa localidade são definidos por quantidades de salários mínimos da região. Escreva um programa para ler do teclado o valor do salário mínimo, **`sm`**, e a quantidade, **`q`**, de salários mínimos que determinada pessoa ganha e, a partir desses dois valores, determinar o salário, **`s`**, da citada pessoa (Obs.: É dado que a multiplicação em Python é indicada pelo operador "**`*`**").

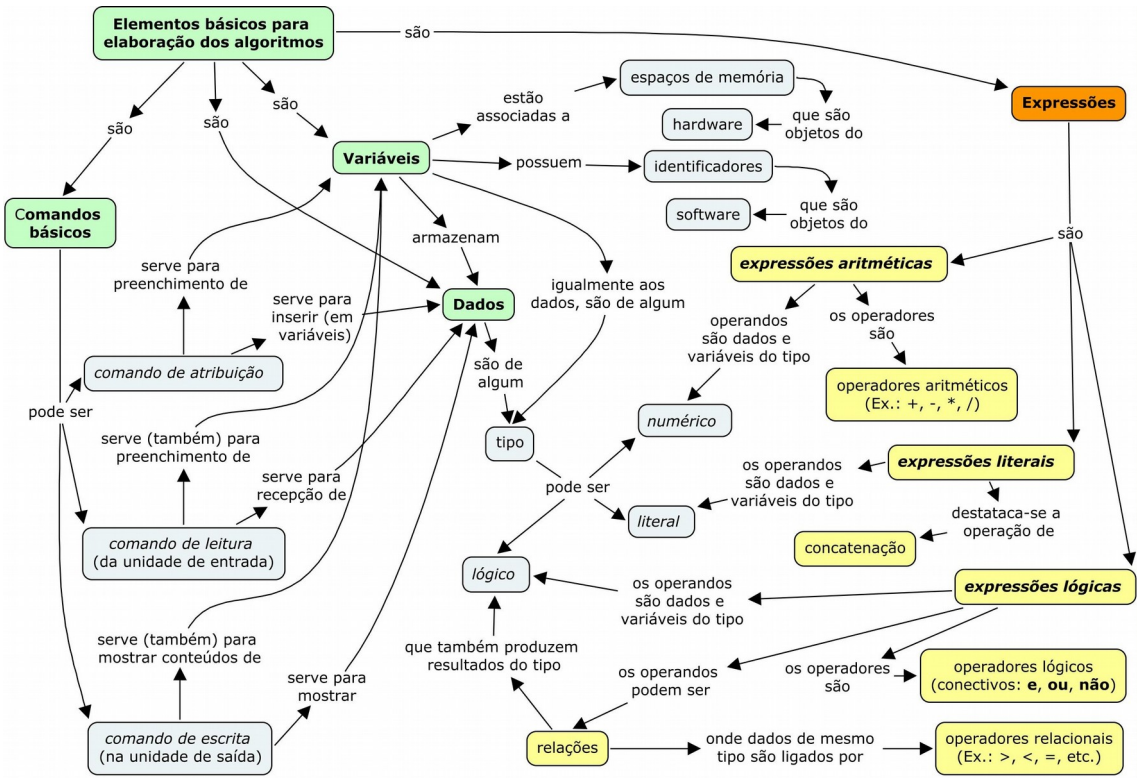
Unidade II.2

Expressões

A palavra "operação" tem muitos significados nas diversas áreas do conhecimento humano. Tomemos o caso das expressões usadas em Matemática como sendo o mais intuitivo do ponto de vista computacional. As operações são efetivadas através de e*xpressões* e são indicadas por símbolos chamados de *operadores*. Estes são aplicados sobre os *operandos*, que são constantes ou conteúdos de variáveis, todos do mesmo tipo.

Esta terminologia amplia seu significado quando admitimos operandos de outros tipos que não sejam somente os numéricos. Então, fazemos uma classificação das expressões de acordo com o tipo dos operandos envolvidos. As expressões, portanto, podem ser *aritméticas* (operandos do tipo numéricos), *lógicas* (operandos do tipo lógico) ou *literais* (operandos do tipo literal). Convém mencionar que as expressões também podem envolver funções que sejam predefinidas na linguagem escolhida para a implementação.

As expressões, juntamente com os elementos abordados na **Unidade 2.1** (Dados, variáveis e comandos básicos), são peças fundamentais para a montagem dos modelos, que são usados para construção dos algoritmos, conforme está exposto no seguinte mapa:



2.2.1 Expressões aritméticas

As expressões aritméticas são as expressões cujos operadores são aritméticos e os operandos são numéricos. Nos algoritmos também podemos usar funções predefinidas, como veremos adiante.

Operadores aritméticos

Estes operadores representam as diversas operações matemáticas de modo semelhante ao que já conhecemos.

Operadores básicos

Os operadores básicos usados nos algoritmos são os de Adição, subtração, multiplicação e divisão, indicados pelos seguintes símbolos:

+ (adição),

- (subtração),

***** (multiplicação) e

/ (divisão)

Exemplo 2.12

Expressões envolvendo apenas +, -, * e /:

```
5.3 + 32.14 - 18*x
2 * (x + y/2)
M/30.0 + 3 * a
(M - N) * (M + N)
a * (m * n + p)
```

Estes símbolos (+, -, * e /) são os mesmos usados nos algoritmos e nas várias linguagens de programação. Isto não acontece com as demais operações. Certas linguagens até são desprovidas de símbolos para algumas delas.

Potenciação e radiciação

Representaremos aqui a potenciação pelo símbolo "^". Não adotaremos operadores para a radiciação (a operação inversa da potenciação), considerando que as linguagens de programação normalmente destinam funções predefinidas para esta tarefa.

Exemplo 2.13

Seja a seguinte questão: Qual é o número `x` que sendo elevado à potência `y` produz o valor `z`?

A solução é conseguida naturalmente efetuando a operação inversa da potenciação que é a radiciação. Isto é, dado que $x^y = z$, então `x` é a raiz de índice `y` de `z`:

$$x = \sqrt[y]{z}.$$

Considerando que não dispomos nesse momento de uma função para o cálculo da raiz, encaminhamos a solução de outra maneira. A solução é a seguinte: Dado que $x^y = z$, sabemos, matematicamente, que $x = z^{1/y}$. Nos algoritmos, escrevemos:

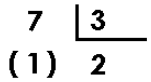
x = z^(1/y).

Obs.: A solução matemática é obtida elevando-se ambos os membros ao expoente **1/y**. Isto é:

$$(x^y)^{1/y} = (z)^{1/y} \Rightarrow x^1 = (z)^{1/y} \Rightarrow x = z^{1/y}$$

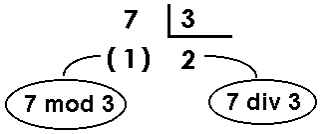
Os operadores `div` e `mod`

Por causa dessa variedade de símbolos, as expressões aritméticas que se usam nos computadores são apenas semelhantes às expressões matemáticas. Para reforçar isto, consideremos a divisão entre dois números inteiros: um, é o dividendo, D, e o outro é o divisor, d. Se q for o quociente e r o resto, aprendemos em Matemática que vale a seguinte relação entre eles: `D = d * q + r`. Por exemplo, se efetuarmos uma divisão inteira de 7 (dividendo) por 3 (divisor), teremos o quociente igual a 2 e o resto igual a 1. Isto é:



Verificamos que 7 = 3 * 2 + 1.

Em Matemática, não há a mágica de obtermos o quociente ou o resto independentemente, sem realizarmos uma divisão inteira. Todavia, pela quantidade de situações requeridas nas soluções algorítmicas, na maioria das linguagens de programação estão definidos operadores que produzem diretamente o quociente e o resto de uma divisão. Nos algoritmos, vamos chamar de `div` o operador para obter o quociente e, de `mod`, o operador para obter o resto. Ou seja, q = D `div` d, e r = D `mod` d. Por exemplo, 7 `div` 3 é igual a 2 e 7 `mod` 3 é igual a 1:



Exemplo 2.14

Estas são algumas das possibilidades de uso dos operadores `div` e `mod`:

a) Dizemos que um número inteiro `A` é divisível pelo inteiro `B` se `A mod B` for igual a 0. Ou seja, se `A` é divisível por `B`, então a divisão de `A` por `B` é exata (de resto igual a zero).

b) O algarismo das dezenas de um número inteiro n de dois dígitos é igual a n `div` 10. Por exemplo, **2**3 = 10 * **2** + 3 (se dividirmos o número 23 por 10, 2 é o quociente e é também o algarismo das dezenas de 23)

c) O algarismo das unidades de um número inteiro n de dois dígitos é igual a n `mod` 10. Por exemplo, 2**3** = 10 * 2 + **3** (se dividirmos o número 23 por 10, 3 é o resto e é também o algarismo das unidades de 23)

d) Dada uma divisão inteira de `x` por `y`, a expressão `x - y * (x div y)` produz o mesmo resultado que x `mod` y. Isto decorre diretamente da relação `D = d * q + r` citada acima, pois `D - d * q = r` (é só mudar a parcela `d * q` para o primeiro membro).

e) Se `k` é o maior número inteiro divisível por `n` e que é menor ou igual a `m`, então `k = n * (m div n)`. Este resultado decorre da divisão inteira de `m` por `n`. Isto é, o produto do divisor (`n`) pelo quociente (calculado por `m div n`) será sempre menor ou igual ao dividendo (m). `k` será igual ao dividendo se o resto for zero e menor, se o resto for diferente de zero.

A solução algorítmica do problema dado no exemplo seguinte põe em prática o caso mencionado no item (e) do exemplo anterior.

Exemplo 2.15

Sejam m e n dois números inteiros. Qual é o maior múltiplo de n que é menor ou igual a m? O algoritmo abaixo lê os dois números (m e n) e responde à pergunta.

```
Algoritmo
1 escrever "Digite dois números inteiros:"
2 escrever "m = "
3 ler m
4 escrever "n = "
5 ler n
6 k ← n * (m div n).
7 escrever "O maior múltiplo de", n,"menor ou igual a", m,"é", k
Fim-Algoritmo
```

Modificação do conteúdo de uma variável

Muitas vezes precisamos trocar o valor de uma variável por outro que seja resultado de uma operação aritmética com o valor antigo. Consideremos o exemplo seguinte.

Exemplo 2.16

Questão: Temos uma variável numérica, `x`, com o valor 5 armazenado e queremos substituí-lo por seu valor acrescido de 1 (um).

A primeira ideia que aparece é fazer o cálculo em separado (em outro local de memória) para substituirmos posteriormente o valor de `x`. Se chamarmos de `aux` o local de memória para guardar temporariamente novo valor de `x`, a mudança é feita da seguinte maneira:

```
x ← 5          (x recebe seu valor original, igual a 5)
aux ← x + 1    (aux recebe o valor antigo de x acrescido de 1)
x ← aux        (x recebe o valor de aux, igual a 6)
```

Aproveitando o significado do comando de atribuição (**Subunidade 2.1.2**), os dois últimos comandos acima podem ser substituídos por apenas um, e a variável auxiliar aux torna-se dispensável:

```
x ← 5          (x recebe seu valor original, igual a 5)
x ← x + 1      (x recebe seu valor antigo acrescido de 1)
```

O sistema calculará primeiramente o valor de x + 1 (com o valor antigo de x) e, logo após, o resultado é atribuído à própria variável x.

O último comando mostrado no ***Exemplo 2.16*** é equivalente a aqueles primeiros que usam a variável auxiliar `aux`. A única diferença é que a definição de um local temporário para o resultado da expressão x + 1 fica a cargo do sistema que implementa estas operações. Do mesmo modo que usamos a adição nesse exemplo, outras operações aritméticas podem ser usadas para modificar conteúdos de variáveis.

Funções predefinidas

As funções predefinidas cobrem, entre outras operações, as trigonométricas e logarítmicas. Pelo caráter particular da coleção de funções disponíveis em cada linguagem, nos algoritmos temos a liberdade de criar representações para as funções de maior interesse. Quando o algoritmo for levado para implementação, as devidas adaptações são feitas. Por exemplo, se usamos a função tangente num algoritmo, é possível que a linguagem de programação escolhida não possua esta função no seu repertório, mas tenha as funções seno e cosseno. Então a adaptação do algoritmo será substituir a chamada de uma tangente pela razão entre os resultados de seno e cosseno (uma propriedade bem conhecida em trigonometria).

No exemplo seguinte, usamos uma função definida como `raizqdr` (*radicando*). Ela existirá apenas nos algoritmos, mas pode ser adaptada à linguagem escolhida no momento da implementação. O uso de uma função predefinida busca apenas agilizar os cálculos e tornar explicita uma operação é bastante frequente, que é o cálculo da raiz quadrada de um número. De fato, estando o operador de potenciação definido, usar esta função é o mesmo que executar uma radiciação com expoente igual a 1/2. Ou seja:

`raizqdr` (*radicando*) = (*radicando*)^(`1/2`).

Exemplo 2.17

Dados os catetos **`b`** e **`c`** de um triângulo retângulo, qual é o valor da hipotenusa **`a`**? A expressão que resolve o problema é a seguinte (pelo Teorema de Pitágoras):

$$a = \sqrt{b^2 + c^2}.$$

Ou seja, `a` é a raiz quadrada de (b^2 + c^2):

```
a = raizqdr(b * b + c * c).
```

O algoritmo seguinte lê do teclado os valores dos catetos b e c e escreve o valor da hipotenusa.

```
Algoritmo
1 escrever "Digite os catetos b e c:"
2 escrever "b = "
3 ler b
4 escrever "c = "
5 ler c
6 a ← raizqdr(b * b + c * c)
7 escrever "Valor da hipotenusa: a =", a
Fim-Algoritmo
```

Exemplo 2.18

Seja a seguinte questão: A que expoente se deve elevar um número `A` para que se obtenha um dado valor B?

Este problema se resume em encontrar `x` tal que $A^x = B$. Aplicando logaritmos (Consulte a bibliografia complementar) em ambos os membros, obtemos $\log(A^x) = \log(B)$ que é equivalente a $x.\log(A) = \log(B)$, de onde extraímos `x`:

$$x = \log(B)/\log(A).$$

O algoritmo abaixo lê os valores `A` e `B` e determina `x` usando a expressão acima:

```
Algoritmo
1 escrever "Digite o valor de A:"
2 ler A
3 escrever "Digite o valor de B:"
4 ler B
5 x ← log(B)/log(A)
6 escrever B, "é resultado de", A, "elevado à potência", x,
Fim-Algoritmo
```

Observação: Existem restrições no algoritmo acima porque `A` e `B` só podem ser positivos. A explicação é a seguinte. Aprendemos que, dado $\log_b A = y$, isto significa que $b^y = A$, pela definição de logaritmo. Portanto, não existe nenhum valor de y que satisfaça esta igualdade para valores de `A` que sejam negativos ou zero, pois `b` é positivo por definição. Algoritmos que atendam a restrições desse tipo serão tratados mais adiante.

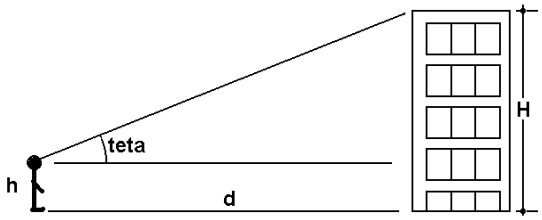
Se tivéssemos ainda que implementar logaritmos, o cálculo mostrado no ***Exemplo 2.18*** não seria viável. Precisaríamos de uma programação avançada que envolveria outra classe de conhecimentos para se conseguir isso. O logaritmo de base 10, por exemplo, já pertence aos pacotes das linguagens de programação.

O exemplo seguinte apresenta a solução algorítmica de um problema de geometria. Observemos que há necessidade de uso de uma função predefinida para o calculo da tangente de um ângulo.

Exemplo 2.19

Um observador de altura `h`, estando a uma distância `d` de um edifício, aponta seu olhar para o topo do mesmo. O ângulo que sua visada faz com o piso (horizontal) é de `teta` graus. Qual é a altura `H` do edifício?

A partir das relações trigonométricas do triângulo retângulo (aqui ele possui o ângulo `teta` conhecido), obtemos que a tangente de `teta` é igual à razão entre o cateto oposto, H – h, e o cateto adjacente, d. Isto é tan(`teta`) = (H – h)/d. Esta expressão permite deduzir que H – h = d.tan(`teta`), ou, H = h + d.tan(`teta`).



O algoritmo seguinte resolve este problema.

```
Algoritmo
1 escrever "Digite:"
2 escrever "A altura do observador(em metros):"
3 ler h
4 escrever "A distância do observador ao prédio(em metros): "
5 ler d
6 escrever "O ângulo de visada do observador (em graus): "
7 ler teta
8 H ← h + d * tan(teta)
9 escrever "A altura do edifício é", H, "metros"
Fim-Algoritmo
```

Laboratório - Expressões aritméticas

Objetivos

Verificar a importância das expressões aritméticas na elaboração dos algoritmos.

Aplicar os conhecimentos da matemática elementar e suas adaptações na elaboração de algoritmos, usando como ferramenta os recursos da linguagem Python.

Recursos e experimentação

Operadores aritméticos básicos. Os operadores aritméticos da linguagem Python são bastante intuitivos e pouco difere do que é implementado na maioria das linguagens.

Começando pelas operações básicas temos os operadores **+**, **-**, *****, **/** (exatamente como vimos nos algoritmos). Porém, em se tratando de linguagem de programação, estas operações envolvem algumas particularidades. É importante observar o seguinte:

- Quando uma expressão tiver pelo menos um valor do tipo `float`, seu resultado também será `float`.

- O resultado da divisão será sempre um `float` (sejam os operandos reais ou inteiros). Especialmente, desejando-se obter apenas a parte inteira do resultado desta divisão deve-se usar o operador "**//**".

Faremos alguns experimentos a seguir usando o interpretador Python interativamente. Digitaremos as expressões e a tecla `ENTER` logo em seguida (o interpretador ficará funcionando como uma calculadora!).

Experimento 01

Usar o interpretador Python para executar as operações seguintes.

```
>>> 6*4 + 34  #Valores inteiros, o resultado será inteiro
58
>>> 6*4.0 + 34  #O resultado será um float pois 4.0 é um float
58.0
>>> 7/3       #Sendo uma divisão o resultado será um float
2.3333333333333335
>>> 7.0/3     #Sendo uma divisão o resultado será um float
2.3333333333333335
>>> 7e0/3     #Sendo uma divisão o resultado será um float
2.3333333333333335
>>> 7//3   #Aqui, deseja-se apenas a parte inteira do result.
2
>>>
```

Observamos no experimento acima que o resultado da expressão **`6*4.0 + 34`** tem com ponto decimal, pois apresenta um operando do tipo `float` (o número **`4.0`**). Os resultados das divisões são sempre números reais. O resultado inteiro será apenas através do operador "**//**". Observa-se também, por causa de limitações na representação numérica interna, que os resultados que são dízimas periódicas sofrem um erro de arredondamento (o período da dízima se destrói no final do número).

Experimento 02

Calcular os valores numéricos das expressões abaixo:

a) `5.3 + 32.14 - 18*x`, para `x = 2.3`;

b) `2 * (x + y/2)`, para `x = 2.3` e `y = 4`;

c) `M/30.0 + 3 * a`, para `M = 25.0` e `a = 5`;

d) `(M - N) / (M + N)`, para `M = 25.0` e `N = 5`;

e) `a * (m * n + p)`, para `a = 5`, `m = 10`, `n = 4` e `p = -15`.

```
>>> x = 2.3
>>> 5.3 + 32.14 - 18*x
-3.960000000000001
>>> y = 4
>>> 2 * (x + y/2)
8.6
>>> M, a = 25.0, 5
>>> M/30.0 + 3 * a
15.833333333333334
>>> N = 5
>>> (M - N) / (M + N)
0.6666666666666666
>>> m, n, p = 10, 4, -15
>>> a * (m * n + p)
125
```

Potenciação e radiciação. A linguagem em estudo possui operador para potenciação, mas não possui operador para radiciação. A potenciação é indicada por dois asteriscos (**). Quanto à radiciação, aplica-se a potenciação com expoente fracionário (conforme foi mostrado no ***Exemplo 2.13***). Ou seja, dado $x^y = z$, se conhecermos `x` e `y`, determinamos `z` por `z = x**y`. E, se conhecermos y e z, determinamos `x` através de `x = z**(1/y)`. No caso particular do cálculo de raiz quadrada, a linguagem Python disponibiliza uma função predefinida como veremos mais adiante.

Experimento 03

Dada a expressão $x^y = z$, qual é o número x que, sendo elevado à quinta potência (y), produz o valor `1024(z)`? (Ou, qual é o valor de `x` tal que $x^5=1024$?) A resposta é a expressão `1024**(1/5)` cujo resultado é:

```
>>> 1024**(1/5)
4.0
```

Os operadores `div` e `mod`. Os operadores `div` e `mod` usados nos algoritmos são implementados da seguinte maneira. O `div` é definido simplesmente como a parte inteira do resultado da divisão entre os operandos. Ou seja, em Python é bastante usar o operador '//'. Já o operador `mod` é representado pelo operador '%'.

Experimento 04

Numa divisão inteira em que o dividendo é 134 e o divisor é 45, qual é o quociente e qual é o resto?

O quociente é obtido por:

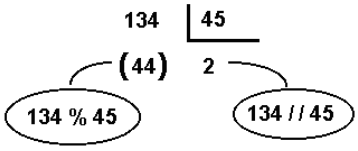
```
>>> 134//45
2
```

`134//45` é a implementação de `134 div 45` (obtém a parte inteira da divisão de 134 por 45)

O resto é obtido por:

```
>>> 134%45
44
```

`134%45` é a implementação de `134 mod 45`. Isto



No experimento seguinte é implementado o algoritmo do ***Exemplo 2.15***.

Experimento 05

O programa abaixo lê dois números, m e n, e responde à pergunta: Qual é o maior múltiplo de n que é menor ou igual a m?

Criar o arquivo `L221_05.py` e digitar as seguintes linhas de código:

```
print ('Digite os números inteiros n e m:')
n = int(input('n = '))
m = int(input('m = '))
k = n * (m//n)
print ('O maior múltiplo de', n, 'menor ou igual a', m,'é', k)
```

Lidos os inteiros `n` e `m`, `m//n` implementa o **`m div n`**, conforme vimos anteriormente.

Execução de `L221_05.py`. Executando o programa e digitando, `m` = 22 e `n` = 3, por exemplo, visualizamos:

```
>>>
Digite os números inteiros m e n:
m = 22
n = 3
O maior múltiplo de n menor ou igual a m é 21
```

De fato, 21 é o maior número divisível por 3 que está dentro da faixa de 0 a 22.

Modificação do conteúdo de uma variável. Para modificar conteúdos armazenados em variáveis procede-se da maneira exposta anteriormente (ver ***Exemplo 2.16***). Por exemplo, se quisermos substituir o valor de uma variável `x` por seu conteúdo acrescido de `1`, podemos escrever em Python:

```
x = x + 1
```

O experimento seguinte implementa a modificação do conteúdo da variável x acima.

Experimento 06

Temos a variável numérica `x` com o valor `5` armazenado. Substituir `x` por seu valor acrescido de `1` (um).

```
>>> x = 5
>>> x = x + 1
>>> print ('Valor atual: x =', x)
Valor atual: x = 6
```

Em particular, a linguagem Python implementa o que podemos chamar de "atribuição composta". Trata-se de uma notação compacta envolvendo o operador em questão (nesse caso, "+") e a atribuição (=). Usando atribuição composta, a expressão acima se torna:

```
x += 1
```

Experimento 07

Temos uma variável numérica, `x`, com o valor 5 armazenado. Substituir x por seu valor acrescido de 1 (um).

```
>>> x = 5
>>> x += 1
>>> print ('Valor atual: x =', x)
Valor atual: x = 6
```

Da mesma maneira que usamos a adição para modificar o conteúdo de uma variável, podemos usar as demais operações básicas e ainda o operador "%" (mod). Ou seja, uma atribuição composta pode ser feita com os operadores:

`+=`, `-=`, `*=`, `/=` ou `%=`

Experimento 08

Calcular e mostrar os novos valores das variáveis i, j, k, m, n, p, q, após a execução das operações abaixo, sabendo-se que seus valores iniciais são: i = j = k = 1, m = n = 21.0 e p = q = 125.

```
j += 5
k *= i+2
m -= n/3
```

```
p /= 2.5
q %= 10
```

Criar o arquivo `L221_08.py` e digitar as seguintes linhas de código:

```
i = j = k = 1
m = n = 21.0
p = q = 125
j += 5
k *= i+2
m -= n/3
p /= 2.5
q %= 10
print ('Valores modificados:')
print ('j =', j)
print ('k =', k)
print ('m =', m)
print ('p =', p)
print ('q =', q)
```

Execução de `L221_08.py`. Executando o programa, obtemos como saída:

```
>>>
Valores modificados:
j = 6
k = 3
m = 14.0
p = 50.0
q = 5
```

Os conteúdos alterados foram os seguintes:

`j += 5` → `j` recebeu seu valor antigo acrescido de `5`: j = 1 + 5 = 6;

`m -= n/3` → `m` com seu valor antigo menos `n/3`: m = 21.0 - 21.0/3 = 21.0 - 7.0 = 14.0

`k *= i+2` → `k` recebeu seu valor antigo multiplicado por `i+2`: k = 1*(1 + 2) = 3;

`p /= 2.5` → `p` com seu valor antigo dividido por `2.5`: p = 125/2.5 = 50.0;

`q %= 10` → `q` recebeu o resto da divisão de seu valor antigo por `10`: q = 125%10 = 5.

Funções predefinidas. Como toda linguagem de programação, Python possui seu conjunto de funções predefinidas. Estas funções estão agrupadas em *módulos* e estes estão reunidos na *biblioteca padrão* da linguagem (trazidas junto com o interpretador).

Funções e constantes para uso nas expressões aritméticas, particularmente, estão reunidas no módulo denominado de **`math`**. Constam na tabela abaixo algumas destas funções:

função.	**Calcula:**

`fabs(x).....`	O valor absoluto de `x`
`pow(x,y).....`	`x**y` (x^y)
`sqrt(x).....`	A raiz quadrada de `x`
`floor(x).....`	O maior inteiro menor ou igual a `x`
`ceil(x).....`	O menor inteiro maior ou igual a `x` (arredonda x por excesso)
`radians(x)...`	O valor em radianos do angulo x dado em graus
`degrees(x)...`	O valor em graus do ângulo x dado em radianos
`sin(x).......`	O seno de `x` (em radianos).
`cos(x).......`	O cosseno de `x` (em radianos)
`tan(x).......`	A tangente de `x` (em radianos)
`acos(x)......`	O arco (em radianos) cujo cosseno é `x`
`asin(x)......`	O arco (em radianos) cujo seno é `x`
`atan(x)......`	O arco (em radianos) cuja tangente é `x`
`exp(x).......`	e^x
`log(x [, b]).`	O logaritmo de `x` na base `b`. Se `b` não for especificado, assume `b = e` (logaritmo natural)
`log10(x).....`	O logaritmo de `x` na base `10` (equivale a `log(x, 10)`)

Constantes predefinidas da biblioteca `math`:

Constante	**Descrição**
`e``.........`	Constante **`e`** = `2.7182818284590451` (Número Nepperiano).
`pi``........`	Constante **π** = `3.1415926535897931`

Para que as funções e constantes de um dado módulo Python sejam utilizadas em um programa, é necessário executar o comando:

`import` **<***módulo***>**

onde **<***módulo***>** indica o módulo Python a ser inserido no programa. A chamada de cada função (*<função_pertencente_ao_módulo>*) deve vir acompanhada do nome do módulo respectivo, da forma

<*módulo***>.***<função_pertencente_ao_módulo>*,

como no exemplo seguinte.

Experimento 09

Este experimento é a implementação da solução algorítmica do ***Exemplo 2.17***: Dados os catetos **`b`** e **`c`** de um triângulo retângulo, qual é o valor da hipotenusa **`a`**?

Criar o arquivo `L221_09a.py` e digitar as seguintes linhas de código:

```
import math
print ('Digite os catetos b e c:')
b = float(input('b = '))
c = float(input('c = '))
a = math.sqrt(b * b + c * c)
print ('Valor da hipotenusa: a =', a)
```

Execução de `L221_09a.py`. Executando o programa e digitando b = 6 e c = 14.4, por exemplo, obtemos como saída:

```
>>>
Digite os catetos b e c:
b = 6
c = 14.4
Valor da hipotenusa: a = 15.6
```

Observamos que a chamada da função `sqrt()` para o cálculo da raiz quadrada é precedida de "`math.`", que é o módulo ao qual esta função pertence. Esta indicação explícita do nome do módulo pode ser referida no início do programa modificando o comando `import` para:

```
from math import *
```

Dessa maneira, o cálculo da hipotenusa `a` pode ser escrito diretamente como:

```
a = sqrt(b * b + c * c)
```

O programa `L221_09b.py` abaixo é uma reescrita do programa `L221_09a.py`:

```
from math import *
print ('Digite os catetos b e c:')
b = float(input('b = '))
c = float(input('c = '))
a = sqrt(b * b + c * c)
print ('Valor da hipotenusa: a =', a)
```

E, considerando os mesmos dados de entrada, a saída será a mesma da versão anterior (é só conferir!).

Experimento 10

O ***Exemplo 2.18*** mostra a solução algorítmica da seguinte questão: A que expoente se deve elevar um número `A` para que se obtenha um dado valor B?

Chamando de `x` o expoente buscado, a implementação daquele algoritmo é a seguinte.

Criar o arquivo `L221_10.py` e digitar as seguintes linhas de código:

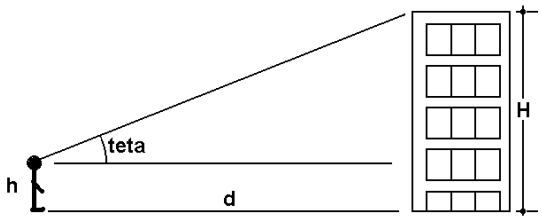
```
from math import *
A = float(input('Digite o valor de A: '))
B = float(input('Digite o valor de B: '))
x = log(B)/log(A)
print (B, 'é o resultado de', A, 'elevado à potência', x)
```

Execução de `L221_10.py`. Se quisermos descobrir a que expoente nós devemos elevar o numero 8 para encontrar o resultado 4096, executamos este programa fornecendo A = 8 e B = 4096 como entrada. Com isso, obtemos a saída:

```
>>>
Digite o valor de A: 8
Digite o valor de B: 4096
4096.0 é o resultado de 8.0 elevado à potência 4.0
>>>
```

Experimento 11

O programa seguinte (`L221_11.py`) é a implementação do algoritmo apresentado no ***Exemplo 2.19***. Ele calcula a altura de um prédio, conhecendo-se: h (altura do observador), d (distância do observador ao prédio, em metros) e teta (ângulo de visada do ob




```
from math import *
print ('Digite:')
h = float(input('A altura do observador(em metros):'))
d = float(input('A distância do obs. ao prédio(em metros):'))
teta = float(input('O ângulo de visada do obs.(em graus):'))
teta = radians(teta)  #converte teta para radianos
H = h + d * tan(teta)
print ('A altura do edifício é de %.3f metros' % H)
```

Nesse programa, foi usada a saída formatada do comando `print` para indicar uma precisão de até milímetros (`1/1000` metro, ou três casas decimais).

Execução de `L221_11.py`. Executando este programa, digitando os seguintes valores de entrada: h = 1.7 m, d = 10 m e teta = 60 graus, encontramos:

```
>>>
Digite:
A altura do observador(em metros):1.7
A distância do obs. ao prédio(em metros):10
O ângulo de visada do obs.(em graus):60
A altura do edifício é de 19.021 metros
>>>
```

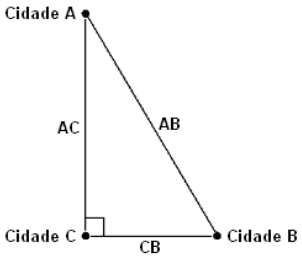
Exercício de autoavaliação

Realize os exercícios abaixo com base nos conhecimentos construídos nesta subunidade. Discuta no fórum dos conteúdos da semana. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Use o interpretador Python interativamente para determinar o valor de `x` na equação: $2^x = 192$, usando apenas um comando (*Sugestão: Aplique as propriedades dos logaritmos*);

2 - O *índice de Massa Corporal* (*IMC*)[4] é um número que ajuda a verificar se uma pessoa adulta está com seu peso normal. A fórmula é `IMC = peso/(altura)`^{2}. A *Organização Mundial de Saúde* (*OMS*) adota faixas de valores para indicar estados considerados normais ou não. Por exemplo, valores do IMC na faixa de 18,5 a 24,9 indicam pessoas de peso normal. Naturalmente, valores acima dessa faixa indicam peso acima do normal e valores abaixo, significam peso abaixo do normal. Escreva um programa para apenas ler o peso e a altura de uma pessoa e, em seguida, calcular e escrever seu IMC.

3 - Os habitantes de uma cidade **A** viajam frequentemente para a cidade **B**. No entanto, esse trajeto obriga que passem também pela cidade **C**. Por isso, planejam construir uma rodovia ligando diretamente as cidades **A** e **B** segundo a disposição mostrada na figura abaixo:



Obs.: Os trajetos conhecidos de AC e CB são retilíneos e formam um ângulo reto entre si. A estrada a ser construída também será retilínea.

Elabore um programa que leia as medidas dos trajetos **AC** e **CB**, calcule e escreva: a) A medida **AB** em km da nova estrada; b) O valor da vantagem em km que a construção da nova estrada trará em relação ao antigo trajeto.

Sugestões: Consulte o **Exemplo 2.17**;Sabe-se também que a soma dos catetos é sempre maior que a hipotenusa.

2.2.2 *Expressões lógicas*

Nas soluções algorítmicas, a necessidade de testes de valores ocorre com frequencia. A partir dos resultados desses testes é que se decide pela execução, ou não, de um determinado bloco de comandos. Citamos as estruturas lógicas na **Subunidade 1.2.2**. Mencionamos a possibilidade de haver condições a serem satisfeitas para que determinado grupo de comandos seja executado. Colocamos agora que as condições são normalmente representadas por *expressões lógicas*.

As expressões lógicas são expressões cujos operadores são *operadores lógicos* e os operandos são *relações*, constantes e/ou variáveis do tipo lógico.

Chamamos de relações as comparações feitas entre duas constantes ou variáveis do mesmo tipo, sendo estas comparações realizadas aplicando operadores denominados de

[4] Consulte: *http://pt.wikipedia.org/wiki/Índice_de_massa_corporal*

operadores relacionais. Os resultados das relações são sempre valores lógicos (verdadeiro ou falso).

Operadores relacionais

As linguagens de programação em geral divergem muito pouco quanto aos símbolos utilizados para estes operadores. Nos algoritmos, podemos escrever normalmente como símbolos matemáticos:

= "igual a",

≠ "diferente de",

< "menor que",

≤ "menor ou igual a",

> "maior que",

≥ "maior ou igual a".

Exemplo 2.20

Dadas as variáveis x e y, as expressões `x = y`, `x ≤ y` e `3*x < y` são relações. Elas terão um valor lógico a assim que atribuirmos valores numéricos a x e a y. Tomemos x = 2 e y = 5. Então:

`x = y produz 2 = 5`, que equivale ao valor lógico `falso`;

`x ≤ y produz 2 ≤ 5`, que equivale ao valor lógico `verdadeiro`;

`3*x < y produz 3*2 < 5 ⇒ 6 < 5`, que equivale ao valor lógico `falso`.

Operadores lógicos

Em Lógica estudam-se as proposições como também as operações lógicas que podem ser realizadas com as mesmas. Tomamos aqui como proposição, uma relação, uma constante, uma variável ou uma expressão mais complexa, desde que signifiquem valores lógicos (ver o tipo lógico, na **Subunidade 2.1.1**). As proposições podem ser combinadas usando os conectivos **e, ou** e **não** produzindo novas proposições:

e (simbolizado por $\wedge$) serve para a *conjunção*. Isto é, dadas duas proposições p_1 e p_2, $p_1 \wedge p_2$ significa que se espera que p_1 e p_2 sejam verdadeiras. Logo, basta uma das proposições ser falsa para que $p_1 \wedge p_2$ também seja falsa.

Exemplo 2.21

São dadas duas proposições, p_1 e p_2. Se p_1 = (5 é ímpar) e p_2 = (4 + 3 > 8) então $p_1 \wedge p_2$ produz o valor `falso` porque, simplesmente, p_2 é falsa.

ou (simbolizado por $\vee$) serve para *disjunção*. Isto é, dadas duas proposições p_1 e p_2, $p_1 \vee p_2$ significa que se espera que pelo menos uma das proposições (p_1 ou p_2) seja verdadeira (é bastante uma das proposições ser verdadeira para que $p_1 \vee p_2$ também seja verdadeira).

Exemplo 2.22

São dadas duas proposições, p_1 e p_2. Se p_1 = (5 é ímpar) e p_2 = (4 + 3 > 8) então $p_1 \vee p_2$ produz o valor `verdadeiro` porque, simplesmente, p_1 é verdadeira.

não (simbolizado por **~**) serve para *negação*. Isto é, dada uma proposição p, **~**p significa o valor oposto ao de p. Logo, se p for verdadeira, **~**p será falsa e, se p for falsa, **~**p será verdadeira.

Exemplo 2.23

São dadas duas proposições, p_1 e p_2. Se p = (5 é ímpar), **~**p produz o valor `falso` (equivale dizer "não é verdade que 5 é impar" e isto é falso). Se p = (4 + 3 > 8), **~**p produz o valor `verdadeiro` (equivale dizer "não é verdade que 4 + 3 > 8" e isto é verdadeiro).

Em resumo, dadas duas proposições p_1 e p_2, podemos escrever:

p_1	p_2	$p_1 \wedge p_2$	$p_1 \vee p_2$	~p_1	~p_2
Verdadeiro	Verdadeiro	Verdadeiro	Verdadeiro	Falso	Falso
Verdadeiro	Falso	Falso	Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Falso	Verdadeiro	Verdadeiro	Falso
Falso	Falso	Falso	Falso	Verdadeiro	Verdadeiro

Os conectivos estão implementados nas linguagens de programação através de operadores lógicos e com eles são construídas as expressões lógicas.

Exemplo 2.24

São dadas as proposições abaixo:

p_1 = (`y = x² + 1`), onde x tem o valor 2 e y tem o valor 5;

p_2 = `var` (onde `var` é uma variável do tipo lógico como valor `verdadeiro`);

p_3 = (`2x - 3 > x`), onde a variável x tem o valor 2;

p_4 = (`10 mod 2 ≠ 0`).

Vemos que p_1 = verdadeiro, p_2 = verdadeiro, p_3 = falso (pois, `2x - 3 > x` resulta em `1 > 2` que é falso) e p_4 = falso (porque o resto da divisão de 10 por 2 é igual zero)

São determinados a seguir os valores de algumas expressões lógicas envolvendo as proposições acima:

a) $p_1 \wedge p_2$ produz o valor verdadeiro. Pois:

$p_1 \wedge p_2 \rightarrow$ (verdadeiro) $\wedge$ (verdadeiro) $\rightarrow$ verdadeiro

b) $p_3 \vee p_4$ produz o valor falso. Pois:

$p_3 \vee p_4 \rightarrow$ (falso) $\vee$ (falso) $\rightarrow$ falso

c) $p_1 \vee (p_2 \wedge p_3)$ produz o valor verdadeiro. Pois:

$p_1 \vee (p_2 \wedge p_3) \rightarrow$ (verdadeiro) $\vee$ ((verdadeiro) $\wedge$ (falso))

$\rightarrow$ (verdadeiro) $\vee$ (falso)

$\rightarrow$ verdadeiro

d) $p_1 \wedge (p_2 \vee p_3)$ produz o valor verdadeiro. Pois:

$p_1 \wedge (p_2 \vee p_3) \rightarrow$ (verdadeiro) $\wedge$ ((verdadeiro) $\vee$ (falso))

$\rightarrow$ (verdadeiro) $\wedge$ (verdadeiro)

$\rightarrow$ verdadeiro

e) $(p_1 \wedge p_3) \wedge (p_2 \vee p_4)$ produz o valor falso. Pois:

$(p_1 \wedge p_3) \wedge (p_2 \vee p_4) \rightarrow$ ((verdadeiro) $\wedge$ (falso)) $\wedge$ ((verdadeiro) $\vee$ (falso))

$\rightarrow$ (falso) $\wedge$ (verdadeiro)

$\rightarrow$ falso

f) $(p_1 \wedge p_2) \wedge \sim(p_3 \vee p_4)$ produz o valor verdadeiro. Pois:

$(p_1 \wedge p_2) \wedge \sim(p_3 \vee p_4) \rightarrow$ ((verdadeiro) $\wedge$ (verdadeiro)) $\wedge \sim$((falso) $\vee$ (falso))

$\rightarrow$ (verdadeiro) $\wedge \sim$(falso)

$\rightarrow$ (verdadeiro) $\wedge$ (verdadeiro)

$\rightarrow$ verdadeiro

O exemplo seguinte mostra o uso de uma expressão lógica com o objetivo de permitir a tomada de uma decisão.

Exemplo 2.25

Desejamos determinar se uma dada equação do 2º. grau ($ax^2 + bx + c = 0$) tem raízes reais ou não, observando apenas seus três coeficientes (a, b, c). Esta decisão pode ser tomada a partir de uma expressão lógica que envolva os três coeficientes da equação dada. A expressão deverá produzir o valor `verdadeiro` se a equação tiver raízes reais e `falso`, no caso contrário.

Para construirmos a expressão citada é suficiente usarmos os nossos conhecimentos básicos de Matemática, da seguinte maneira. Sabemos que uma equação do segundo grau tem raízes reais se o discriminante, $b^2 - 4ac$, que pertence à fórmula de Baskara (para resolução de equações polinomiais do 2º. grau), não for negativo. Logo, podemos escrever:

$$b^2 - 4ac \geq 0$$

Assim, se forem dados os coeficientes a, b e c, o resultado da expressão lógica $b^2-4ac \geq 0$) indicará que a equação tem raízes reais se seu valor for `verdadeiro` e indicará que não tem raizes reais se seu valor for `falso`.

Laboratório - Expressões lógicas

Objetivos

Verificar a importância das expressões lógicas na elaboração dos algoritmos.

Aplicar elementos de Lógica na elaboração de algoritmos e conferir utilizando a linguagem Python.

Recursos e experimentação

Operadores relacionais. Os operadores relacionais da linguagem Python são:

"==" (igual a), "!=" (diferente de),

"<" (menor que), "<=" (menor ou igual a),

">" (maior que), ">=" (maior ou igual a).

Experimento 01

Usando o interpretador Python interativamente, criar as variáveis `x`, `y` com os valores `x = 2`, `y = 5`, e verificar os valores relações `x <= y`, `x == y`, `3*x < y` (ver ***Exemplo 2.20***). Para obter o valor da expressão é suficiente digitar a expressão desejada e a tecla `ENTER` em seguida.

```
>>> x, y = 2, 5
>>> x == y
False
>>> x <= y
True
>>> 3*x < y
False
```

Operadores lógicos. Os operadores lógicos da linguagem Python são **and** (correspondente ao **e**, $\wedge$), **or** (correspondente ao **ou**, $\vee$) e **not** (correspondente ao **não**, **~**).

O experimento seguinte tem objetivo de verificar o que foi apresentado no ***Exemplo 2.24***.

Experimento 02

Criar as variáveis `x = 2`, `y = 5` e `var = True` e ainda `p1 = (y == x**2 + 1)`, `p2 = var`, `p3 = (2*x - 3 > x)`, `p4 = (10 % 2 != 0)`. A partir destas variáveis, obter os resultados das seguintes expressões lógicas: (`p1 and p2`), (`p3 or p4`), (`p1 or (p2 and p3)`), (`p1 and (p2 or p3)`), (`((p1 and p3) and (p2 or p4))`) e (`((p1 and p2) and not(p3 or p4))`).

```
>>> x, y = 2, 5
>>> var = True
>>> p1 = (y == x**2 + 1)
>>> p2 = var
>>> p3 = (2*x - 3 > x)
>>> p4 = (10 % 2 != 0)
>>> print ('p1 =',p1,' p2 =',p2,' p3 =',p3,' p4 =',p4)
p1 = True  p2 = True  p3 = False  p4 = False
>>> p1 and p2
True
>>> p3 or p4
```

```
False
>>> p1 or (p2 and p3)
True
>>> p1 and (p2 or p3)
True
>>> (p1 and p3) and (p2 or p4)
False
>>> (p1 and p2) and not(p3 or p4)
True
```

O experimento seguinte implementa a expressão lógica elaborada no ***Exemplo 2.25***. A citada expressão está associada à decisão sobre o tipo das raízes de uma dada equação do segundo grau.

Experimento 03

Atribuir valores aos coeficientes a, b e c da equação $ax^2 + bx + c = 0$ e escrever o valor lógico `True` se a equação tiver raízes reais e o valor `False`, no caso contrário.

Usando o interpretador Python interativamente, vamos atribuir os valores `a = 1.0`, `b = -5.0`, e `c = 6.0` e escrever o valor da expressão lógica.

```
>>> a, b, c = 1., -5., 6.
>>> print ('Tem raízes reais...', (b*b - 4*a*c>=0))
Tem raízes reais... True
```

Isto é, o resultado do experimento mostra que a equação $x^2 - 5x + 6 = 0$ tem raízes reais, embora estas raízes ainda não tenham sido calculadas.

Exercício de autoavaliação

Verifique seus conhecimentos construídos nesta subunidade. Realize os exercícios abaixo usando o interpretador Python interativamente. Discuta no fórum dos conteúdos da semana. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Elabore uma expressão lógica envolvendo as medidas de altura (em metro) e peso (em kg) de uma pessoa, que produza o valor `True` se esta pessoa estiver com peso considerado normal e `False` no caso contrário (consulte o item 2 dos exercícios da **Subunidade 2.2.1**).

2 - Elabore uma expressão lógica envolvendo um dado número inteiro positivo, que produza o valor True se o tal número for um múltiplo de 17 e `False` no caso contrário.

3 - Elabore uma expressão lógica envolvendo um dado caractere, que produza o valor `True` se o tal caractere for a letra 's' (maiúscula ou minúscula) e `False` no caso contrário.

2.2.3 Expressões literais

Os dados do tipo literal são tratados pelas linguagens de programação como tipos *estruturados*. Os dados organizados por estruturas facilitam a manipulação de várias posições de memória utilizando um mesmo nome, mas este é um tema que veremos mais adiante, no **Módulo IV**. Geralmente as linguagens dispõem também de um conjunto de funções predefinidas para facilitar a manipulação.

A operação básica com valores literais é a concatenação de cadeias de caracteres, e o operador literal de concatenação é indicado pelo sinal "+". Concatenar duas cadeias significa "colar" uma na outra formando uma única cadeia de caracteres.

Sem tratar dos detalhes agora, podemos dizer que construir uma expressão literal consiste em combinar duas ou mais cadeias de caracteres, usando operador de concatenação e, eventualmente, funções predefinidas. O resultado da expressão é uma nova cadeia.

Exemplo 2.26

Dadas as variáveis do tipo literal `nome1` e `nome2`, onde `nome1` contém "Jose" e `nome2` contém "Santos", a expressão `nome1+ " dos " + nome2` tem como resultado o valor literal `"Jose dos Santos"`.

Laboratório - Expressões literais

Objetivos

Verificar o uso de expressões literais nos algoritmos.

Aplicar a concatenação de literais e casos particulares da linguagem Python.

Recursos e experimentação

Em Python a operação básica com valores do tipo `string` também é a concatenação e o operador de concatenação é indicado pelo sinal "+". O experimento seguinte serve para confirmar o resultado da expressão dada no ***Exemplo 2.26***.

Experimento 01

Criar as variáveis `nome1` e `nome2` com os respectivos valores "Jose" e "Santos", e escrever o resultado da expressão: `nome1+ " dos " + nome2`. Usando o interpretador Python interativamente,

```
>>> nome1 = 'José'
>>> nome2 = 'Santos'
>>> print (nome1 + ' dos ' + nome2)
Jose dos Santos
```

Em particular, a linguagem Python permite a operação de produto de uma string por um número inteiro. O operador é * (asterisco). O símbolo é idêntico ao aritmético, mas o efeito

não é. O resultado é uma nova cadeia formada pela concatenação do operando string, repetido quantas vezes for indicado pelo operando inteiro. Por exemplo, a expressão `'a ' * 3` tem como resultado a `string` `'a a a '`. O produto `'b ' * 2` equivale à `string` `'b b '`. Então, a expressão `'a ' * 3 + 'b ' * 2` produzirá a string `'a a a b b '`.

Experimento 02

Escrever no monitor os resultados das seguintes expressões: (`'a ' * 3`), (`'b ' * 2`) e (`'a ' * 3 + 'b ' * 2`).

```
>>> print ('a ' * 3)
a a a
>>> print ('b ' * 2)
b b
>>> print ('a ' * 3 + 'b ' * 2)
a a a b b
```

Exercício de autoavaliação

Realize os exercícios abaixo com base nos conhecimentos construídos nesta subunidade.

1 - Interativamente, crie duas variáveis numéricas, x e y, fazendo x = 5 e y = 4.3 e execute o comando abaixo:

print ('A média aritmética entre '+ str(x) + ' e ' + str(y) + ' é ' + str((x+y)/2.))

Tente explicar o resultado encontrado.

2 - Crie as `strings` `s1 = 'Alagoas'` e `s2 = 'Brasil'`. Use o interpretador Python para executar o comando abaixo e explique o resultado:

```
print (2*'-' + s1 + 3*'-' + s2 + 2*'-')
```

3 - Em Python, podemos converter valor lógico em numérico. Isto é, converter `True` em `1` e `False` em `0` (veja o **item 5** dos **Exercícios** da **Subunidade 2.1.2**). Então, este fato, aliado ao de podermos multiplicar uma *string* por um número, sugere uma interessante aplicação. Por exemplo, vamos tomar os coeficientes `a`, `b` e `c` de uma equação do segundo grau ($ax^2 + bx + c = 0$), conforme a abordagem do ***Exemplo 2.25*** e o ***Experimento 03*** do laboratório da **Subunidade 2.2.2**. Se a expressão lógica $b^2 - 4ac \geq 0$ resultar em verdadeiro (ou, True), é porque a equação tem raízes reais. Se a referida expressão produzir False (falso) significa que a equação não tem raízes reais. Criemos então a variável `coef`:

```
coef = int(b**2 - 4*a*c >= 0)
```

Logo, `coef` receberá o valor 1 se (`b**2 - 4*a*c` $\geq$ `0`) for `True` e `0`, se a mesma for `False`.

Assim, o resultado do experimento referido logo acima pode ser modificado de modo que a resposta seja '`Sim`' quando a equação tiver raízes reais:

```
>>> a, b, c = 1., -5., 6.
```

```
>>> coef = int(b**2 - 4*a*c >= 0)
>>> print ('Tem raízes reais...', coef*'Sim')
Tem raízes reais... Sim
>>>
```

Notemos que o resultado do produto **`coef*'Sim'`** será `'Sim'` quando `coef` for 1 e será `''` (*string* vazia) quando `coef` for 0 (zero). Então, faça uma melhoria nesse resultado, de modo que a resposta possa ser `'Não'` quando a equação não tiver raízes reais.

4 - Aproveitando os conhecimentos adquiridos na questão anterior, faça um programa para ler os coeficientes `a`, `b` e `c`, de uma equação do segundo grau e, em seguida, escrever uma frase com o seguinte formato:

"A equação dada `'tem'`(ou, `'não tem'`) *raízes reais."*

O programa escolherá uma das strings, 'tem' ou 'não tem', conforme a equação, respect., tenha ou não tenha raízes reais.

MÓDULO III

Estruturas de controle

Foi visto, já no primeiro módulo deste curso, que os algoritmos estruturados organizam-se sob estruturas lógicas denominadas de *estruturas de controle*. Classificamos então em *estrutura sequencial*, *estruturas de seleção* e *estruturas de repetição*. No segundo módulo, aplicamos os casos de uso da estrutura sequencial para apresentar os elementos básicos dos algoritmos. Detalharemos neste terceiro módulo as estruturas de seleção e as de repetição. Como a própria terminologia sugere, as estruturas de seleção *selecionam* blocos de comandos a serem executados de acordo com determinadas restrições, e as de repetição *executam repetidamente* blocos de comandos observando-se determinados critérios para esta repetição.

Objetivos

- Identificar as estruturas de seleção e de as estruturas de repetição como estruturas de controle básicas;
- Construir as estruturas de seleção e de repetição segundo uma notação algorítmica.
- Aplicar adequadamente as estruturas de seleção e de repetição na montagem das soluções dos problemas;

Unidades

- Unidade III.1 - Estruturas de Seleção
- Unidade III.2 - Estruturas de repetição

Unidade III.1

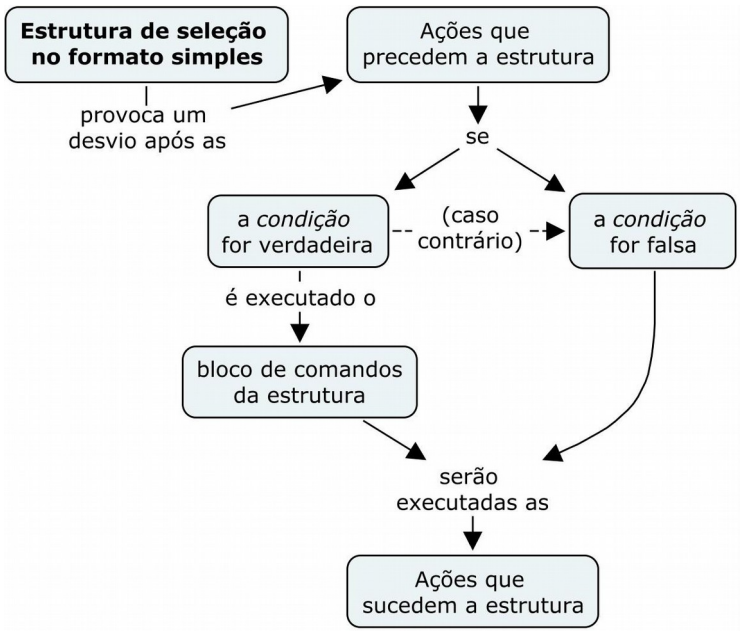
Estruturas de Seleção

De acordo com o que já expusemos, uma estrutura de seleção é usada para controlar logicamente a execução de comandos. Ou seja, um bloco de comandos somente é executado se uma determinada condição for satisfeita. Essa estrutura pode assumir dois formatos. Um, é chamado de *formato simples*, *seleção simples* ou, na prática, *estrutura "se-então"*. O outro é chamado de *formato composto*, *seleção composta*, ou *estrutura "se-então-senão"*.

3.1.1 Formato "se - então"

Usamos uma estrutura de seleção no seu formato simples quando precisamos condicionar a execução de apenas um bloco de comandos. Isto é, um bloco de comandos será executado se uma dada condição for verdadeira e nenhum outro comando especificamente estará amarrado ao caso desta condição ser falsa. Em termos de escrita dos algoritmos, após o teste da condição, se esta for falsa, o controle da estrutura simplesmente prosseguirá em seu curso, seguindo os demais passos.

Podemos descrever esta estrutura através do seguinte mapa:



Sintaxe

Esse formato da estrutura de seleção pode ser descrito nos algoritmos com a seguinte sintaxe:

```
se (condição) então:
    bloco de comandos
```

Onde: A **`condição`** é dada por um valor lógico proveniente de uma expressão lógica, uma constante ou uma variável do tipo lógico.

O **`bloco de comandos`** somente será executado se a condição for verdadeira.

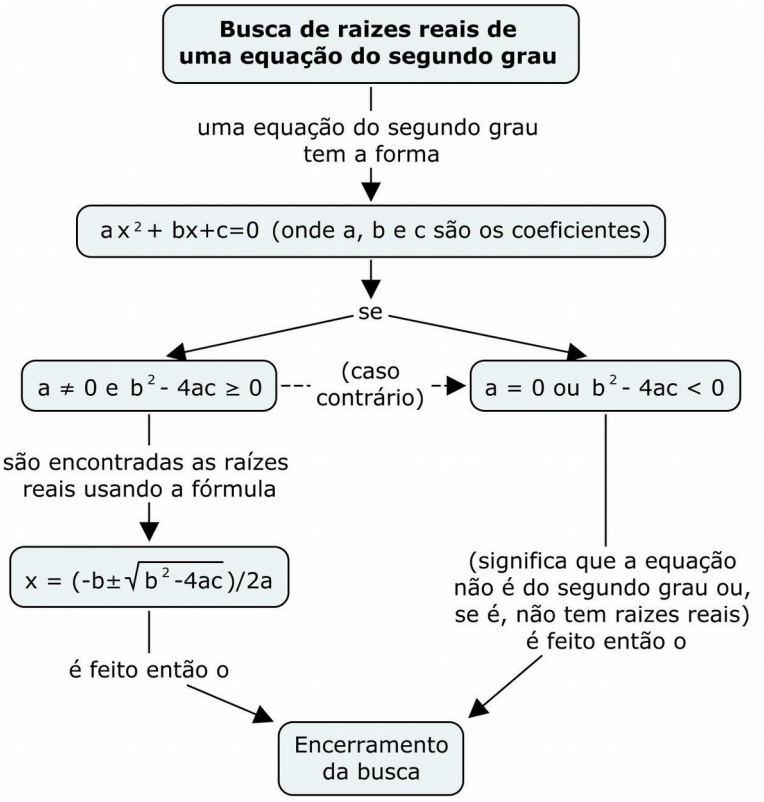
Convém reforçar que, no caso de a condição ser falsa, essa estrutura é indiferente. Isto é, não tem nenhuma ação prevista para ser realizada se a condição não for verdadeira. Simplesmente o curso do algoritmo será continuado com o próximo comando que existir após a estrutura (esta é uma maneira de simplificar a codificação).

Por exemplo, vamos considerar que desejamos resolver uma equação do segundo grau somente se esta tiver raízes reais, e nada faremos se a mesma tiver raízes complexas. O algoritmo do exemplo seguinte resolve esse problema.

Exemplo 3.1

Problema: Determinar as raízes reais de uma equação do segundo grau ($ax^2 + bx + c = 0$, sendo $a \neq 0$), dados os seus coeficientes `a`, `b` e `c`.

Solução: A solução do problema pode ser determinada a partir do seguinte mapa, que mostra o uso da fórmula de Baskara na busca das raízes reais da equação.



O algoritmo seguinte traduz em ações os conceitos acima, escrevendo as raízes `x1` e `x2` se estas forem números reais. Observação: Ressaltamos aqui a boa prática da endentação. Os comandos amarrados à estrutura são recuados para não deixar dúvidas de que dependem da condição.

```
Algoritmo
1 escrever "Forneça os coeficientes a, b e c da equação:"
2 ler a
3 ler b
4 ler c
5 se (a ≠ 0) e (b^2 - 4*a*c ≥ 0) então:
5.1   x1 ← (-b + raizqdr(b^2 - 4*a*c))/(2*a)
5.2   x2 ← (-b - raizqdr(b^2 - 4*a*c))/(2*a)
5.3   escrever "Raízes da equação:", x1,"e",x2
Fim-Algoritmo
```

Com certeza, no exemplo acima, o usuário não digitaria propositalmente o valor zero para o coeficiente de x^2 porque, em sendo equação do segundo grau, então o termo ax^2 tem que existir (ou seja, $a \neq 0$). Por outro lado, observamos que existe divisão por `2*a` nas expressões dos comandos 5.1 e 5.2. Logo, para não haver divisão por zero, o valor `a` não pode ser zero. Em resumo, os comandos 5.1 (o cálculo de x1), 5.2 (o cálculo de x2) e 5.3 (a escrita de x1 e x2) serão executados se a equação for de fato do segundo grau (`a ≠ 0`) e tiver raízes reais (`b^2 - 4*a*c ≥ 0`).

Vejamos no quadro abaixo um teste com dados numéricos do algoritmo acima. Podemos chamar esse processo de "rastreamento" do algoritmo, no caso, para resolver a equação $\mathbf{x^2 - 5x + 6 = 0}$. Os dados de entrada são, portanto, **a** = **1**, **b** = **-5** e **c** = **6**.

Passos do algoritmo:	Var. **a**	Var. **b**	Var. **c**	É verdade que `(a ≠ 0)` e `(b^2-4*a*c ≥ 0)`?	Var. **x1**	Var. **x2**	*Observações:*
1º. passo	-	-	-	-	-	-	Escreve a mensagem: "Forneça os coef..."
2º. passo	1	-	-	-	-	-	Leitura de **a**
3º. passo		-5	-	-	-	-	Leitura de **b**
4º. passo			6	-	-	-	Leitura de **c**
5º. passo				`(1 ≠ 0)` e `(-5)^2 - 4*1*6 ≥ 0` (isto é verdadeiro). Então, vai calcular e escrever x1 e x2.	**3**	**2**	(5.1) Calcula **x1** (5.2) Calcula **x2** (5.3) Escreve o resultado: "Raízes da..."
Estado final da memória	1	-5	6		**3**	**2**	

No quadro, no passo 5.1, o cálculo de x1, é `(-(-5) + raizqdr((-5)^2 - 4*1*6))/2*1` que produz o valor `3`. O cálculo de x2, no passo 5.2, é `(-(-5) - raizqdr((-5)^2 - 4*1*6))/2*1` que produz o valor `2`. Portanto, sendo as raízes iguais a 3 e 2, respectivamente, a mensagem escrita ao final `será: "Raízes da equação: 3 e 2"`

No laboratório logo a seguir teremos a oportunidade de repetir esse teste e verificar o algoritmo para novos valores.

Laboratório - Estrutura de Seleção "se - então"

Objetivos

Verificar o caso básico de aplicação da estrutura de seleção "se - então".

Conferir os algoritmos estruturados apresentados nesta subunidade através da visualização dos resultados após a implementação na linguagem Python.

Recursos e experimentação

Vimos que as estruturas de seleção são montadas para controlar a execução de seus blocos de comandos. A linguagem Python implementa todas as estruturas de seleção (simples e compostas). Tomemos então o caso da estrutura simples.

Sintaxe. Em Python a estrutura de seleção simples **"se - então"** ganha o seguinte formato (realmente simples!):

```
if condição:
    bloco de comandos
```

onde os comandos do *bloco de comandos* (a serem executados sob a *condição*) são escritos com o mesmo recuo (endentação) e a *condição* pode vir sem parênteses.

Convém destacar a maneira como a marcação do bloco de comandos se torna explícita na implementação. As linguagens de programação normalmente usam delimitadores para deixar bem claro quais são os comandos que serão executados quando uma condição for satisfeita, ou não. Por exemplo, na linguagem C os blocos comandos ficam dentro de chaves, "`{}`" e, em Pascal, ficam entre os delimitadores "`begin`" (marcando o início do bloco) e "`end`" (marcando o fim do bloco), como observamos no **Módulo I**, na **Subunidade 1.2.3** (que trata da implementação dos algoritmos). Porém, em Python, obedecer à endentação é o bastante! Ou seja, o recuo na edição do programa indica que comandos pertencem a que estruturas, e não é apenas algo feito somente para melhorar a legibilidade dos programas. Logo, na programação em Python a endentação se torna obrigatória.

Faremos então, a seguir, a implementação do algoritmo dado no ***Exemplo 3.1***. O citado algoritmo usa uma estrutura de seleção simples para verificar se a equação do segundo grau tem raízes reais antes de determinar estas raízes.

Experimento 01

O programa seguinte determina as raízes reais de uma equação do segundo grau, dados os seus coeficientes `a`, `b` e `c`. O programa somente escreverá as raízes (`x1` e `x2`) se a equação for de fato do segundo grau ($a \neq 0$) e se as raízes forem números reais.

Criar o arquivo `L311_01.py` e digitar as seguintes linhas de código:

```
from math import*
print ('Forneça os coeficientes da equação:')
```

```
a = float(input('a = '))
b = float(input('b = '))
c = float(input('c = '))
if (a != 0) and (b**2 - 4*a*c >=0):
    x1 = (-b + sqrt(b**2 - 4*a*c))/(2*a)
    x2 = (-b - sqrt(b**2 - 4*a*c))/(2*a)
    print ('Raízes da equação:', x1,'e',x2)
```

Observamos que os três comandos submetidos à estrutura **`if`** estão com a mesma endentação. Se isto não for feito ocorrerá um erro.

Execução de `L311_01.py`. Executando o programa e digitando `a` = 1, `b` = -5 e `c` = 6 (para resolver a equação $x^2 - 5x + 6 = 0$), visualizamos:

```
>>>
Forneça os coeficientes da equação:
a = 1
b = -5
c = 6
Raízes da equação: 3.0 e 2.0
>>>
```

Experimentando agora com os valores `a` = 3, `b` = -5 e `c` = 6 (tentando resolver a equação $3x^2 - 5x + 6 = 0$), obtemos:

```
>>>
Forneça os coeficientes da equação:
a = 3
b = -5
c = 6
>>>
```

Ou seja, após a leitura dos dados, nada acontece! A explicação é a seguinte: Não há comandos previstos para serem executados quando a condição (`(a != 0) and (b**2 - 4*a*c >=0)`) for falsa e isto se verificará toda vez que o coeficiente `a` for igual a zero ou o discriminante for negativo (que é justamente o caso, pois: $-5^2 - 4.3.6 = 25 - 72 = -47$).

Exercício de autoavaliação

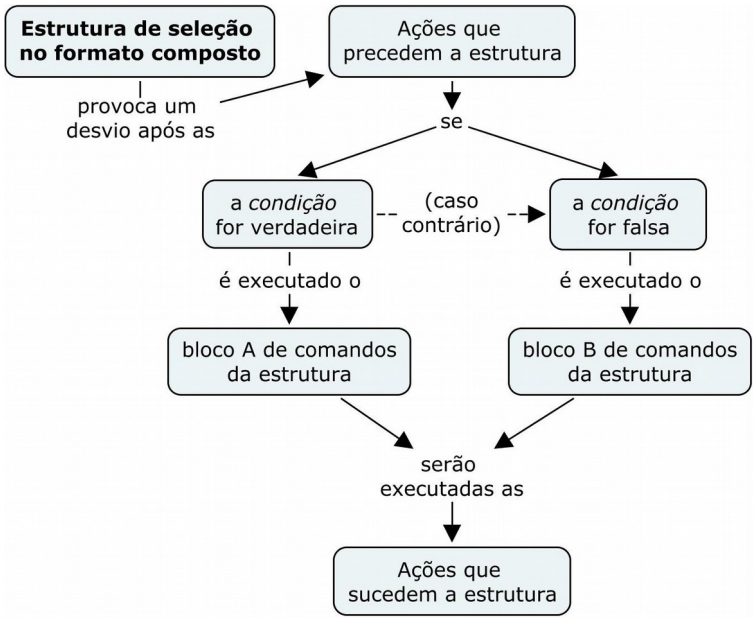
Resolva os exercícios abaixo com base nos conhecimentos construídos nesta subunidade. Experimente previamente e discuta em seguida no fórum dos conteúdos da semana. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Elabore um programa em Python para ler do teclado o raio R de uma circunferência, calcular e escrever o comprimento da mesma (cuja fórmula é $2\pi R$) somente se foi digitado um número positivo para o valor de R.

2 - Elabore um programa em Python que lê dois números inteiros, **a** e **b**, e mostra os resultados de **a+b**, **a-b**, **a*b** e, se **a** e **b** forem positivos, escreve o quociente e o resto da divisão de **a** por **b**.

3.1.2 Formato "se - então - senão"

Quando estivermos prevendo alguma ação para o caso de a condição da estrutura de seleção ser falsa, o formato composto deverá ser usado. Podemos descrever o citado formato utilizando o seguinte mapa:



Sintaxe

Podemos representar esta estrutura nos algoritmos usando a sintaxe seguinte:

```
se (condição) então:
    bloco de comandos A
senão:
    bloco de comandos B
```

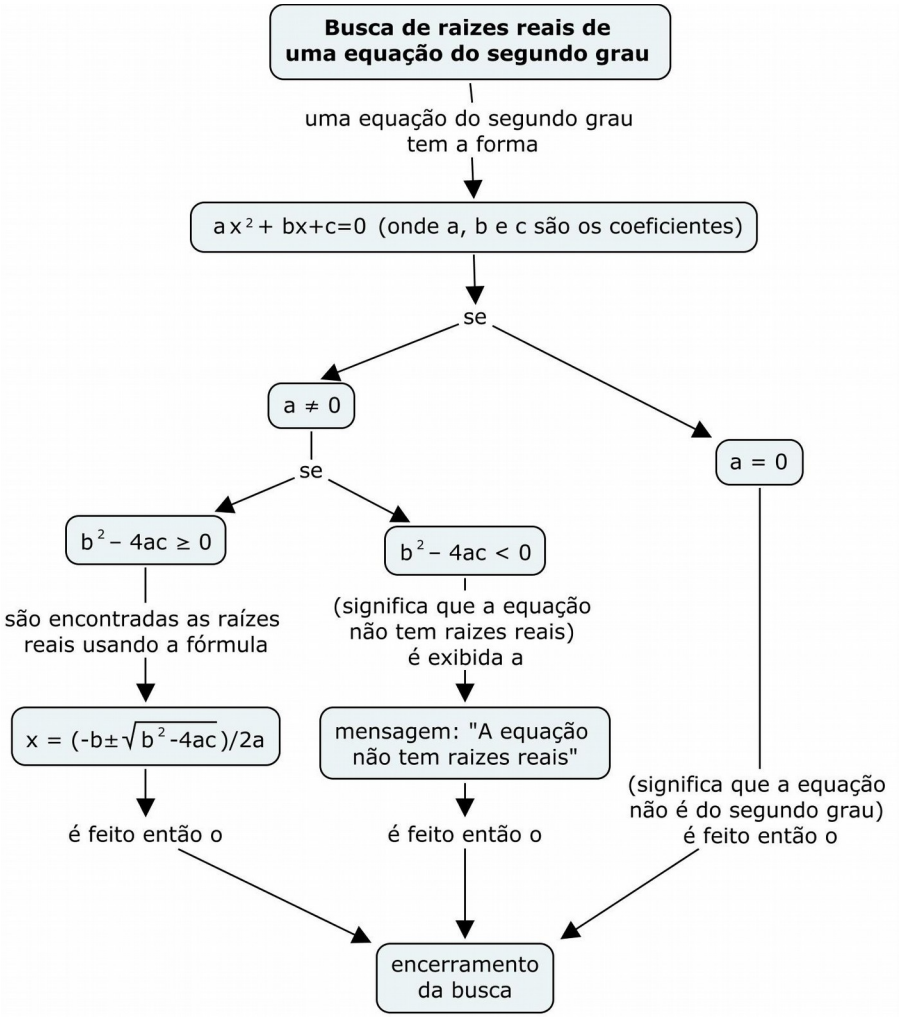
Onde: Dada a **condição** da estrutura, o **bloco de comandos A** somente será executado se a condição for verdadeira e o **bloco de comandos B** será executado somente se a condição for falsa. Ou seja, é sempre certo que um dos blocos de comandos será executado exclusivamente. E, após esta execução, o fluxo de controle do algoritmo continuará normalmente a partir do comando seguinte à estrutura.

Usando a seleção composta, podemos incrementar o algoritmo do ***Exemplo 3.1***, acrescentando ações como, por exemplo, a emissão mensagens. Na versão abaixo, uma mensagem é emitida no caso de a equação não ter raízes reais.

Exemplo 3.2

Problema: Determinar as raízes reais de uma equação do segundo grau ($ax^2 + bx + c = 0$, sendo $a \neq 0$), dados os seus coeficientes `a`, `b` e `c`. Se a equação não tiver raízes reais, será emitida apenas a mensagem `"A equação não tem raízes reais"`.

A solução agora deve prever uma mensagem quando a equação não tiver raízes reais, como está mostrado no mapa abaixo:



A solução algorítmica passa a ser:

```
Algoritmo
1 escrever "Forneça os coeficientes a, b e c da equação:"
2 ler a
3 ler b
4 ler c
5 se (a ≠ 0) então:
5.1   se (b^2 - 4*a*c ≥ 0) então:
5.1.1       x1 ← (-b + raizqdr(b^2 - 4*a*c))/(2*a)
5.1.2       x2 ← (-b - raizqdr(b^2 - 4*a*c))/(2*a)
```

```
5.1.3       escrever "Raízes da equação:", x1,"e",x2
      senão:
5.1.4       escrever "A equação não tem raízes reais"
Fim-Algoritmo
```

Nesse exemplo, há o interesse específico em enviar uma mensagem relativa somente à discussão das raízes (Não há mensagens para o caso de o coeficiente `a` ser igual a zero). No passo 5, consta somente uma seleção simples para verificar a condição `a ≠ 0` (O coeficiente `a` é diferente de zero?). O bloco de comandos dessa estrutura é apenas o passo 5.1, que, por sua vez, é uma estrutura de seleção composta.

Essa última estrutura (o passo 5.1) executa os comandos 5.1.1, 5.1.2 e 5.1.3 (calcula e escreve as raízes) se for verdade que a equação tem raízes reais (isto é, se `b^2 - 4*a*c ≥ 0`) e, no caso contrário (isto é, se `b^2 - 4*a*c < 0`, que é o contrário de positivo ou zero), ela executa o comando 5.1.4 (emite uma mensagem informando que a equação não tem raízes reais).

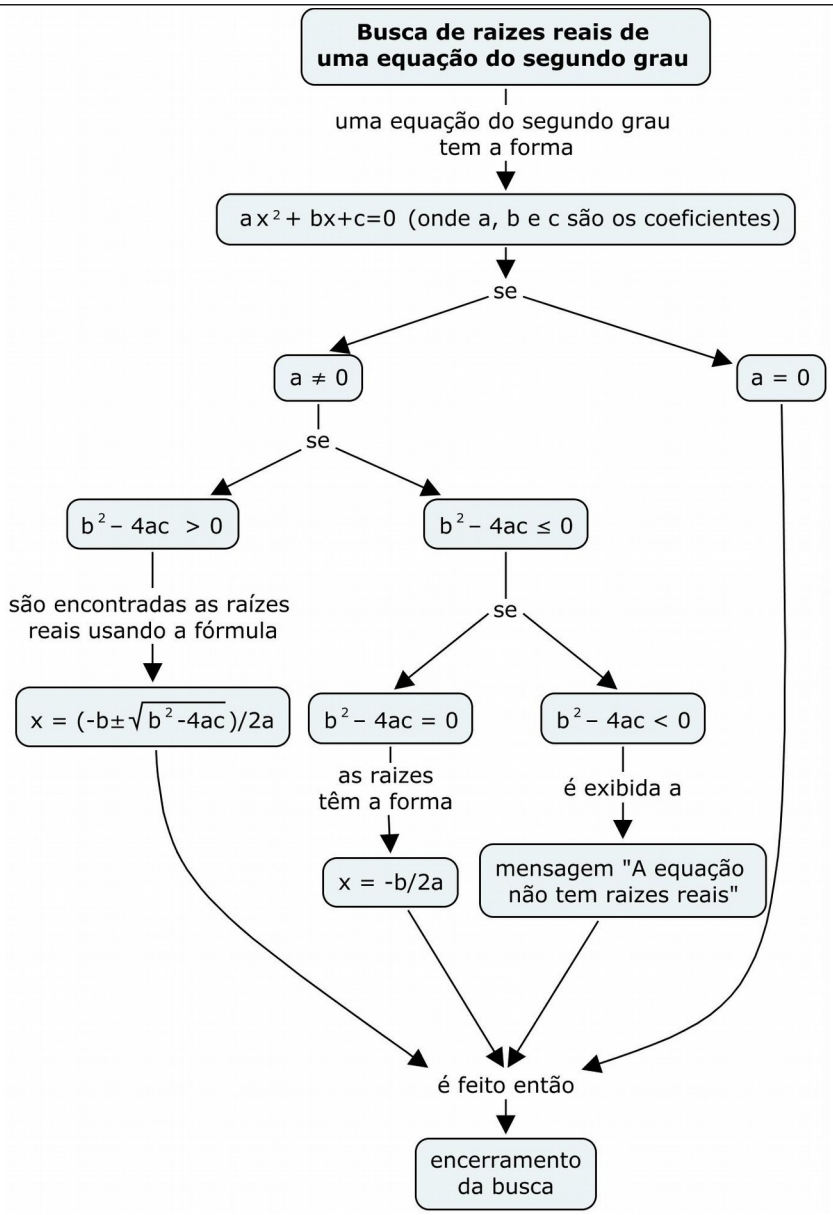
O ***Exemplo 3.2*** serve também para indicar que não há restrições quanto ao uso de estruturas mergulhadas em blocos de comandos de outras (vemos que os passos 5 e 5.1 são estruturas de seleção, e 5.1 está submetido à estrutura 5).

Vamos tomar mais uma vez a resolução da equação mostrada nos ***Exemplos 3.1*** e ***3.2***. Se quisermos incrementar um pouco mais, por exemplo, informando quando a equação tiver raízes iguais, uma estrutura de seleção composta deverá ser inserida. Vejamos o ***Exemplo 3.3*** a seguir.

Exemplo 3.3

Problema: Determinar as raízes reais de uma equação do segundo grau ($ax^2 + bx + c = 0$, sendo $a \neq 0$), dados os seus coeficientes `a`, `b` e `c`. Se a equação não tiver raízes reais, será emitida apenas a mensagem `"A equação não tem raízes reais"` e, se a equação tiver raízes reais iguais apenas um valor será calculado e isto será indicado na mensagem.

O mapa seguinte expressa a solução do problema, contemplando o caso de a equação ter raízes iguais.



A solução algorítmica correspondente é:

```
Algoritmo
1 escrever "Forneça os coeficientes a, b e c da equação:"
2 ler a
3 ler b
4 ler c
5 se (a ≠ 0) então:
5.1   se (b^2 - 4*a*c > 0) então:
5.1.1       x1 ← (-b + raizqdr(b^2 - 4*a*c))/(2*a)
5.1.2       x2 ← (-b - raizqdr(b^2 - 4*a*c))/(2*a)
5.1.3       escrever "Raízes da equação:", x1,"e",x2
      senão:
5.1.4       se (b^2 - 4*a*c = 0) então:
```

```
                escrever "A equação tem duas raízes iguais a", -b/(2*a)
            senão:
                escrever "A equação não tem raízes reais"
Fim-Algoritmo
```

Observamos que a condição `b^2 - 4*a*c ≥ 0`, presente nas soluções anteriores, foi agora separada em duas expressões e destinadas a condições de estruturas diferentes. Isto se explica porque, anteriormente, ter raízes reais era o suficiente. Desta vez, as duas possibilidades estão contempladas separadamente. Ou seja, sendo duas raízes reais, estas podem ser diferentes (se `b^2 - 4*a*c > 0`), ou iguais (se `b^2 - 4*a*c = 0`).

Portanto, a estrutura 5.1 executará os comandos 5.1.1, 5.1.2, 5.1.3 se as raízes da equação forem reais e diferentes, e executará o comando 5.1.4 no caso contrário. Segue-se, então, que o caso contrário de se ter raízes reais e diferentes é ter raízes que não são reais ou não são diferentes. Por isso, o passo 5.1.4 é uma estrutura de seleção justamente para decidir isto. Como restam apenas estas duas últimas possibilidades, a estrutura verifica se a equação tem raízes iguais. Se isto for verdade ela escreve o valor da raiz (usando uma fórmula simplificada) e, se não for verdade, é emitida apenas a mensagem de que a equação não tem raízes reais.

Vimos nos exemplos anteriores que as estruturas de seleção podem ser usadas progressivamente à medida que variamos as restrições para a execução de comandos. Podemos admitir que, se o problema for de escolha, não haverá como escapar do uso de estruturas de seleção. Todavia, muitas vezes, a seleção é intrínseca ao problema diferentemente dos exemplos acima, onde a parte central do problema é o cálculo das raízes fundamentado na fórmula de Baskara.

O problema seguinte (***Exemplo 3.4***) pode ser considerado clássico no sentido de que o bom uso das estruturas é que resolve o problema.

Exemplo 3.4

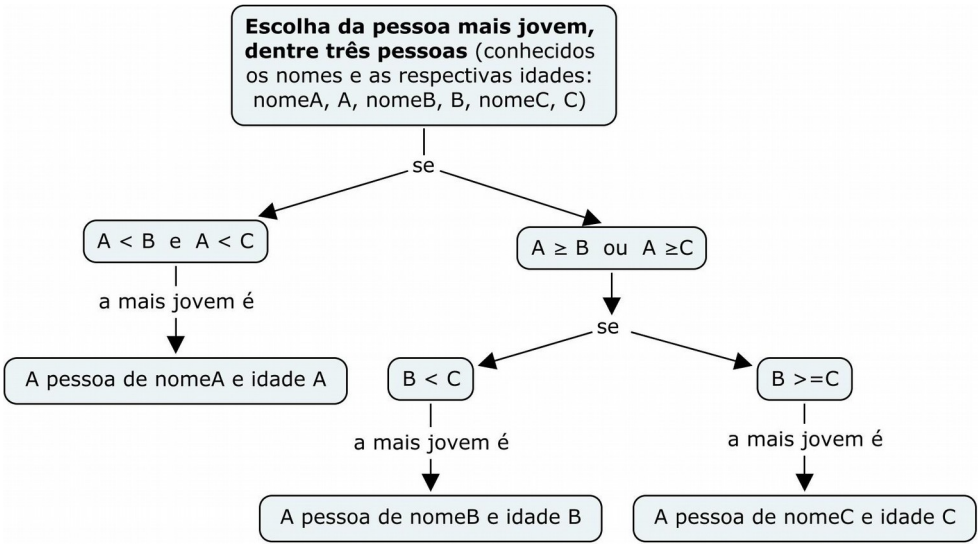
Consideremos o seguinte problema: Dados os nomes e as respectivas idades de três pessoas nascidas em anos distintos, mostrar os dados da pessoa mais jovem.

De antemão, sabemos que essa tarefa é bastante simples para uma pessoa com conhecimento mínimo de números. Porém, nosso interesse aqui é demonstrar que, para resolvermos esse problema, precisamos fazer um pequeno jogo de comparações muito semelhante ao que faríamos mentalmente.

Sem o uso do computador, buscaríamos o menor valor para idade numa ordem, por hábito, da esquerda para a direita. O primeiro número da esquerda é comparado com os demais. Se este for o menor de todos, não precisaremos continuar o processo.

Ora, se este nosso primeiro "golpe" não der certo, então o menor só poderá ser um dos dois restantes. Pronto, agora fica fácil: comparamos os dois últimos e o menor deles será o menor de todos!

Lembrando que estes números são idades de pessoas e que sabemos seus nomes, a saída do problema é simplesmente escrever o menor número e o respectivo nome associado, como mostra o mapa seguinte:



A correspondente solução algorítmica é apresentada abaixo:

```
Algoritmo
1 escrever "Fornecer os nomes e as respect. idades das pessoas:"
2 ler nomeA
3 ler A
4 ler nomeB
5 ler B
6 ler nomeC
7 ler C
8 se (A < B) e (A < C) então:
      escrever "Com",A,"anos,",nomeA,"é a pessoa mais jovem"
  senão:
      se (B < C) então:
            escrever "Com",B,"anos,",nomeB,"é a pessoa mais jovem"
      senão:
            escrever "Com",C,"anos,",nomeC,"é a pessoa mais jovem"
Fim-Algoritmo
```

Observação: A consideração apenas de idades distintas é somente uma simplificação do problema. O funcionamento do algoritmo depende disto. Se forem digitados valores iguais, o algoritmo escreverá os dados da última pessoa como sendo a mais jovem.

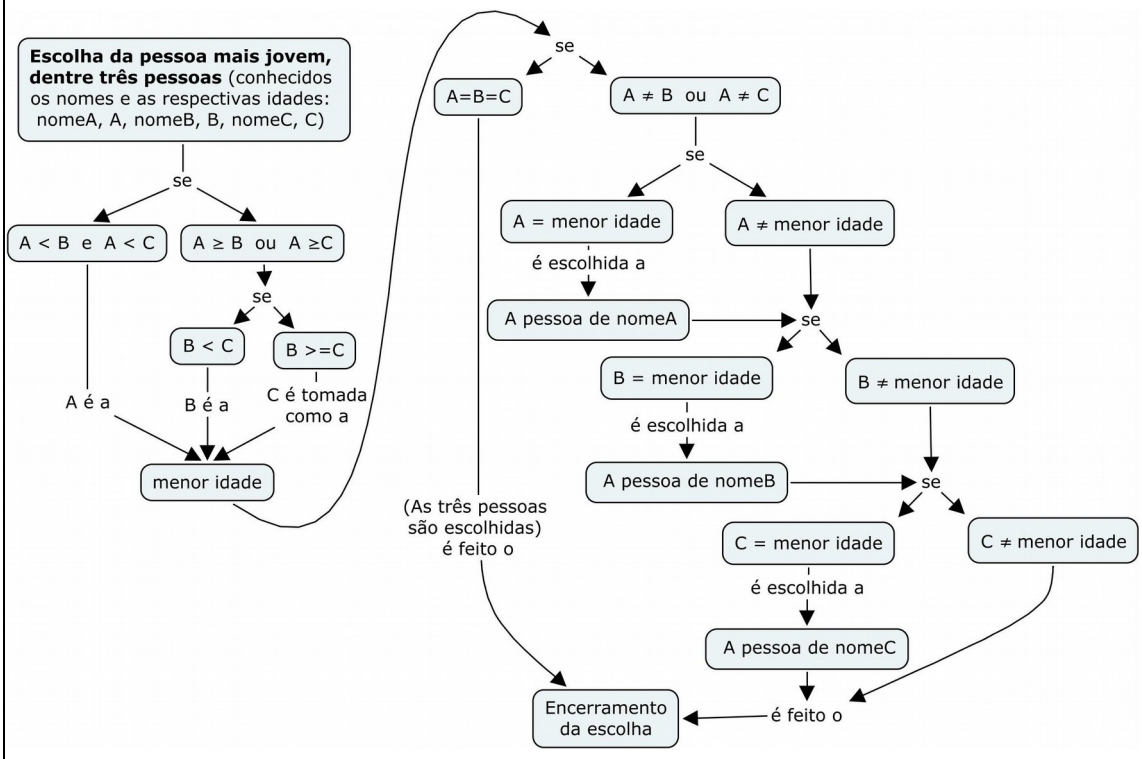
No algoritmo dado, as leituras dos nomes e das respectivas idades das pessoas ocorrem do primeiro ao sétimo passo. No oitavo passo, o algoritmo verifica se já a primeira pessoa (cuja idade é `A`) é a mais jovem. O teste é: `A` é menor que `B` e menor que `C`? Se for verdadeiro, o algoritmo emite a mensagem adequada e se encerra. Mas, se for falso (isto é, se a primeira pessoa não for a mais jovem), é necessário fazer mais um teste: `B` é menor que `C`? Se for verdadeiro, é porque a segunda pessoa (cuja idade é `B`) é a mais jovem. Caso contrario, resta somente a terceira pessoa (cuja idade é `C`) como sendo a mais jovem.

O problema do ***Exemplo 3.4*** não contempla o caso de pessoas com a mesma idade. Isto não se constitui uma falha da solução porque faz parte do enunciado, explicitamente, que as idades a serem fornecidas pelo usuário devem ser distintas. Vamos então avaliar que alterações no algoritmo devem ser feitas para adaptá-lo a um enunciado que não faça restrições de idade (e somente que sejam números inteiros e positivos). Tomemos então o problema do exemplo seguinte.

Exemplo 3.5

Problema: Dados os nomes e as respectivas idades de três pessoas, mostrar a menor idade e o(s) nome(s) da(s) pessoa(s) com esta idade.

A solução começa pela determinação da menor idade. Somente assim poderemos verificar em seguida que pessoas tem esta idade. Isto é o que está apresentado no mapa seguinte:



Na correspondente solução algorítmica é mostrada abaixo:

```
Algoritmo
1 escrever "Fornecer os nomes e as respect. idades das pessoas:"
2 ler nomeA
3 ler A
4 ler nomeB
5 ler B
6 ler nomeC
7 ler C
8 se (A < B) e (A < C) então:
      menorIdade ← A
  senão:
      se (B < C) então:
            menorIdade ← B
      senão:
            menorIdade ← C
9 se (A = B) e (A = C) então:
9.1   escrever "As três pessoas tem",menorIdade,"anos de idade"
  senão:
9.2   escrever "Pessoa(s) mais nova(s), com",menorIdade,"anos:"
9.3   se (A = menorIdade) então:
            escrever nomeA
9.4   se (B = menorIdade) então:
            escrever nomeB
9.5   se (C = menorIdade) então:
            escrever nomeC
Fim-Algoritmo
```

Após a leitura dos dados das pessoas (nos passos de 1 a 7), o algoritmo seleciona, no passo 8, a menor idade e a armazena na variável `menorIdade`. O processo é semelhante ao executado no passo 8 do algoritmo do ***Exemplo 3.4***, exceto que naquele exemplo o valor da menor idade é imediatamente escrito e não é armazenado. A estrutura do passo 9 verifica primeiramente se todas as pessoas tem a mesma idade. Se isto acontecer é porque todas elas possuem a idade que foi armazenada na variável `menorIdade` e o comando 9.1 é executado, mostrando esta informação. Não sendo todas idades iguais, os comandos 9.2 a 9.5 serão executados, mas existe ainda a possibilidade de igualdade duas a duas. Para simplificar a emissão de mensagens, algoritmo faz a opção por escrever uma relação onde pode constar dois nomes, se existir duplas de pessoas com a mesma idade, ou apenas um, se as idades forem distintas. A relação de nomes é escrita através das estruturas dos passos 9.3, 9.4 e 9.5. Observemos que as estruturas são independentes, ou seja, se existir mais de uma pessoa com idade igual à `menorIdade` terá seu nome escrito.

Laboratório - Estrutura de Seleção "se - então - senão"

Objetivos

Verificar o caso básico de aplicação da estrutura de seleção "se - então - senão".

Conferir os algoritmos estruturados apresentados nesta subunidade através da visualização dos resultados após a implementação na linguagem Python.

Recursos e experimentação

Sintaxe. Em Python, a estrutura de seleção composta **"se - então - senão"** ganha o seguinte formato:

```
if condição:
    bloco de comandos A
else:
    bloco de comandos B
```

onde a *condição*, o *bloco de comandos A* e o *bloco de comandos B* obedecem as mesmas regras de sintaxe da estrutura simples.

Vejamos então a implementação do algoritmo dado no ***Exemplo 3.2***.

Experimento 01

O programa seguinte determina as raízes reais de uma equação do segundo grau, dados os seus coeficientes `a`, `b` e `c`. O programa escreverá as raízes (`x1` e `x2`) se a equação for de fato do segundo grau ($a \neq 0$) e se as raízes forem números reais. Em sendo uma equação do segundo grau, no caso de não haver raízes reais, o programa emitirá a mensagem: `"A equação não tem raízes reais"`.

Criar o arquivo `L312_01.py` e digitar as seguintes linhas de código:

```
from math import*
print ('Forneça os coeficientes da equação:')
a = float(input('a = '))
b = float(input('b = '))
c = float(input('c = '))
if a != 0:
    if b**2 - 4*a*c >=0:
        x1 = (-b + sqrt(b**2 - 4*a*c))/(2*a)
        x2 = (-b - sqrt(b**2 - 4*a*c))/(2*a)
        print ('Raízes da equação:', x1,'e',x2)
    else:
        print ('A equação não tem raízes reais')
```

Execução de `L312_01.py`. Executando o programa e digitando `a` = 3, `b` = -5 e `c` = 6 (tentando resolver a equação $3x^2 - 5x + 6 = 0$), visualizamos:

```
>>>
Forneça os coeficientes da equação:
a = 3
b = -5
c = 6
A equação não tem raízes reais
>>>
```

Este resultado mostra uma melhor informação para o usuário, em comparação com o obtido com a execução do programa `L311_01.py`. Experimentemos mais uma vez, resolvendo a equação: $2x^2 - 12x + 18 = 0$. Executando o programa e digitando `a` = 2, `b` = -12 e `c` = 18, obtemos:

```
>>>
Forneça os coeficientes da equação:
a = 2
b = -12
c = 18
Raízes da equação: 3.0 e 3.0
>>>
```

O experimento seguinte tem o objetivo de realizar a implementação do ***Exemplo 3.3***.

Experimento 02

O programa abaixo escreve as raízes reais de uma equação do segundo grau. O detalhe em relação à versão testada no experimento anterior é que, havendo raízes iguais, o programa apresentará uma mensagem diferenciada.

Criar o arquivo `L312_02.py` e digitar as seguintes linhas de código:

```
from math import*
print ('Forneça os coeficientes da equação:')
a = float(input('a = '))
b = float(input('b = '))
c = float(input('c = '))
if a != 0:
    if b**2 - 4*a*c > 0:
        x1 = (-b + sqrt(b**2 - 4*a*c))/(2*a)
        x2 = (-b - sqrt(b**2 - 4*a*c))/(2*a)
        print ('Raízes da equação:', x1,'e',x2)
    else:
```

```
        if b**2 - 4*a*c == 0:
            print ('A equação tem raízes iguais a', -b/(2.*a))
        else:
            print ('A equação não tem raízes reais')
```

Execução de `L312_02.py`. Usaremos este programa para resolver a equação: $2x^2 - 12x + 18 = 0$. Executando e digitando `a` = 2, `b` = -12 e `c` = 18, visualizamos:

```
>>>
Forneça os coeficientes da equação:
a = 2
b = -12
c = 18
A equação tem raízes iguais a 3.0
>>>
```

Experimento 03

O programa seguinte é a implementação do algoritmo dado no ***Exemplo 3.4***. O programa lê os nomes e as idades (distintas) de três pessoas e escreve os dados da pessoa mais jovem.

Criar o arquivo `L312_03.py` e digitar as seguintes linhas de código:

```
print ('Fornecer os nomes e as respectivas idades das pessoas:')
nomeA = input('1a.pessoa, nome: ' )
A = int(input('          idade: '))
nomeB = input('2a.pessoa, nome: ' )
B = int(input('          idade: '))
nomeC = input('3a.pessoa, nome: ' )
C = int(input('          idade: '))
if (A < B) and (A < C):
    print ('Com',A,'anos,',nomeA,'é a pessoa mais jovem.')
else:
    if (B < C):
        print ('Com',B,'anos,',nomeB,'é pessoa a mais jovem.')
    else:
        print ('Com',C,'anos,',nomeC,'é a pessoa mais jovem.')
```

Execução de `L312_03.py`. Executando o programa e digitando os seguintes nomes e idades: Maria com 25 anos, José com 23 anos e João com 37 anos, obtemos:

```
>>>
Fornecer os nomes e as respectivas idades das pessoas:
1a.pessoa, nome: Maria
          idade: 25
```

```
2a.pessoa, nome: José
          idade: 23
3a.pessoa, nome: João
          idade: 37
Com 23 anos, José é pessoa a mais jovem.
>>>
```

Logo a seguir, no **Experimento 04**, teremos a oportunidade de verificar o algoritmo do ***Exemplo 3.5*** (com estes dados e com outros em que existam pessoas com a mesma idade).

Experimento 04

O programa abaixo implementa do algoritmo do ***Exemplo 3.5***.

Criar o arquivo `L312_04.py` e digitar as seguintes linhas de código:

```
print ('Fornecer os nomes e as respectivas idades das pessoas:')
nomeA = input('1a.pessoa, nome: ' )
A = int(input('          idade: '))
nomeB = input('2a.pessoa, nome: ' )
B = int(input('          idade: '))
nomeC = input('3a.pessoa, nome: ' )
C = int(input('          idade: '))
if (A < B) and (A < C):
    menorIdade = A
else:
    if (B < C):
        menorIdade = B
    else:
        menorIdade = C
if A==B and A==C:
    print ('As três pessoas tem',menorIdade,'anos'    )
else:
    print ('Pessoa(s) mais nova(s), com',menorIdade,'anos:')
    if A==menorIdade:
        print (nomeA)
    if B==menorIdade:
        print (nomeB)
    if C==menorIdade:
        print (nomeC)
```

Observação: Muitas vezes, os blocos de comandos das estruturas, na verdade, reduzem-se a apenas um comando. Quando isto ocorre, a linguagem Python permite que o comando seja

colocado logo adiante do sinal “:”. Usando essa permissão da linguagem, o programa `L312_04.py` ganha nova feição, todavia mantendo a funcionalidade:

```
print ('Fornecer os nomes e as respectivas idades das pessoas:')
nomeA = input('1a.pessoa, nome: ' )
A = int(input('          idade: '))
nomeB = input('2a.pessoa, nome: ' )
B = int(input('          idade: '))
nomeC = input('3a.pessoa, nome: ' )
C = int(input('          idade: '))
# Determina a menorIdade idade:
if (A < B) and (A < C): menorIdade = A
else:
    if (B < C): menorIdade = B
    else: menorIdade = C
# Mostra o resultado:
if A==B and A==C: print('As três pessoas tem',menorIdade,'anos')
else:
    print ('Pessoa(s) mais nova(s), com',menorIdade,'anos:')
    if A==menorIdade: print (nomeA)
    if B==menorIdade: print (nomeB)
    if C==menorIdade: print (nomeC)
```

Execução de `L312_04.py`. Executando o programa e entrando com os mesmos dados do experimento anterior, obtemos:

```
>>>
Fornecer os nomes e as respectivas idades das pessoas:
1a.pessoa, nome: Maria
          idade: 25
2a.pessoa, nome: José
          idade: 23
3a.pessoa, nome: João
          idade: 37
Pessoa(s) mais nova(s), com 23 anos:
José
>>>
```

Vamos agora experimentar com os dados: Ana com 23 anos, José com 23 anos e João com 37 anos:

```
>>>
Fornecer os nomes e as respectivas idades das pessoas:
```

```
1a.pessoa, nome: Ana
          idade: 23
2a.pessoa, nome: José
          idade: 23
3a.pessoa, nome: João
          idade: 37
Pessoa(s) mais nova(s), com 23 anos:
Ana
José
>>>
```

Vejamos então quando todos tiverem a mesma idade, digitando os nomes Maria, Pedro e Antônio, todos com 25 anos de idade:

```
>>>
Fornecer os nomes e as respectivas idades das pessoas:
1a.pessoa, nome: Maria
          idade: 25
2a.pessoa, nome: Pedro
          idade: 25
3a.pessoa, nome: Antônio
          idade: 25
As três pessoas tem 25 anos
>>>
```

Exercício de autoavaliação

Elabore os programas abaixo com base nos conhecimentos construídos nesta subunidade. Experimente previamente e discuta em seguida no fórum dos conteúdos da semana.

1 - Escreva uma nova versão do programa `L312_02.py` de modo que o mesmo emita a mensagem "A equação não é do 2o. grau" quando for digitado um coeficiente **a** igual a zero.

2 - No quarto item do exercício de autoavaliação da **Subunidade 1.2.2**, consta o seguinte problema: Dada a altura (em metros) de uma pessoa e seu sexo (1-masculino ou 2-feminino), calcular e escrever seu peso ideal p (em kg), utilizando as fórmulas empíricas:

p = 72.7*altura - 58.0, se for homem e

p = 62.1*altura - 44.7, se for mulher.

Implemente na linguagem Python o algoritmo elaborado naquele exercício.

3.1.3 Formato encadeado

Nos exemplos acima observamos estruturas de seleção que se encaixam para produzir a solução do problema. Estendendo essa ideia, podemos permitir isto para uma sequência contendo uma quantidade qualquer de estruturas de seleção.

O encadeamento de estruturas de seleção é usado geralmente quando se tem uma lista de condições a serem atendidas. São usadas quantas estruturas forem necessárias.

Sintaxe

Quando o número de estruturas aumenta o encadeamento pode receber uma nova endentação, conforme está indicado abaixo:

```
se (condição 1) então:
    bloco de comandos 1
senão se (condição 2) então:
    bloco de comandos 2
...
senão se (condição n) então:
    bloco de comandos n
senão:
    bloco de comandos n+1
```

Onde: Dada uma lista de condições (**condição 1**, **condição 2**, **condição 3**, ..., **condição n**), e os blocos de comandos **1**, **2**, **3**, .., **n**, **n+1**, o **bloco de comandos 1** será executado somente se a **condição 1** for verdadeira, o **bloco 2** será executado somente se a **condição 2** for verdadeira, e assim por diante. Isto se verificará até o **bloco de comandos n** porque o **bloco n+1** (o último) somente será executado se nenhuma das condições dadas for verdadeira. Este último bloco é facultativo, ou seja, a última estrutura pode ser uma seleção simples.

Observação: Devido ao encontro dos termos "senão", da estrutura anterior, e "se", da estrutura seguinte, o encadeamento de estruturas muitas vezes recebe o nome de estrutura "senão-se".

Vamos usar este formato no exemplo a seguir para resolver um problema considerado típico para o uso de estruturas de seleção encadeadas. O tema é o mesmo abordado no item 2 do exercício de autoavaliação da **Subunidade 2.2.1** (Expressões aritméticas). Trata-se do cálculo do IMC (índice de Massa Corporal) de uma pessoa adulta.

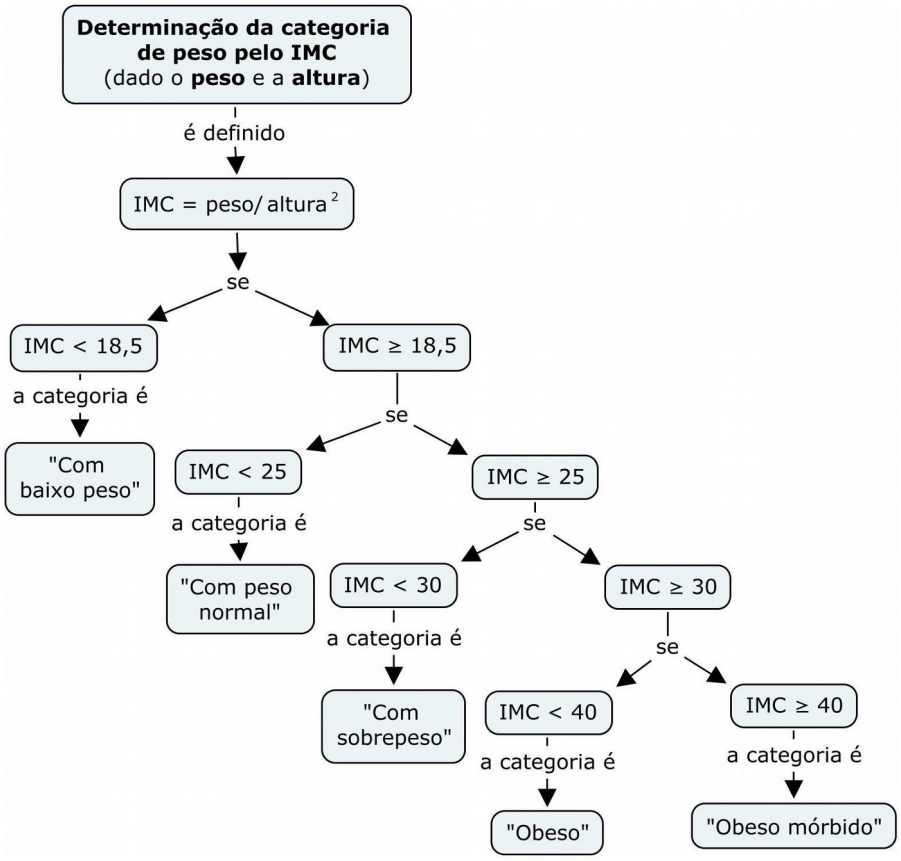
Exemplo 3.6

Como já sabemos, o índice de Massa Corporal (IMC) é um número calculado pela fórmula $IMC = peso/(altura)^2$, dados o peso e a altura de uma pessoa adulta. A

Organização Mundial de Saúde usa esse índice para categorizar as condições de peso das pessoas, conforme está relacionado abaixo:

IMC em adultos	**Categoria**
Menor que 18,5.....	Com baixo peso
Maior ou igual a 18,5 e menor que 25....	Com peso normal
Maior ou igual a 25 e menor que 30.....	Com sobrepeso
Maior ou igual a 30 e menor que 40.....	Obeso
A partir de 40.....	Obeso mórbido

A determinação da categoria de peso pelo IMC pode ser expressa pelo seguinte mapa:



O algoritmo seguinte mostra a categoria de peso de uma pessoa, a partir do seu peso e sua altura.

```
Algoritmo
1 escrever "Digite o peso(kg):"
2 ler peso
3 escrever "Digite a altura(m):"
4 ler altura
5 IMC ← peso/(altura^2)
6 se (IMC < 18.5) então:
      escrever "Categoria: Com baixo peso"
  senão se (IMC < 25) então:
```

```
      escrever "Categoria: Com peso normal"
  senão se (IMC < 30) então:
      escrever "Categoria: Com sobrepeso"
  senão se (IMC < 40) então:
      escrever "Categoria: Obeso"
  senão:
      escrever "Categoria: Obeso mórbido"
Fim-Algoritmo
```

Nesse algoritmo, os passos de 1 a 4 são dedicados à leitura dos dados de entrada: os passos 1 e 3 são solicitações ao usuário e os passos 2 e 4 são as leituras propriamente. O cálculo do IMC é o quinto passo. Para este cálculo, devemos observar que há uma restrição quanto ao valor da altura: a altura não pode ser igual a zero. Na verdade, o algoritmo não está protegendo quanto a enganos do usuário, mas, por enquanto, vamos deixar assim. O objetivo (didático) é destacar o passo 6 que é o encadeamento de estruturas de seleção.

Examinando encadeamento do exemplo acima, podemos ver que o último comando será executado somente se o IMC calculado não se encaixar em nenhuma das faixas anteriores. Considerando que as faixas anteriores cobrem os valores do IMC menores que 40, o último comando tornou-se indispensável porque corresponde a uma das categorias citadas no problema, referida ao IMC maior ou igual a 40.

Laboratório - Encadeamento de Estruturas de Seleção

Objetivos

Conferir o uso do encadeamento de estruturas de seleção quando múltiplas condições tiverem que ser verificadas.

Usar os recursos disponíveis na linguagem Python para aplicar os conceitos e testar os algoritmos desta subunidade.

Recursos e experimentação

A linguagem Python possui um formato especial para a sua estrutura de seleção quando implementa o encadeamento destas estruturas. Como vimos nesta subunidade, o encadeamento de estruturas faz com que o `"senão"` da estrutura anterior encontre o `"se"` da estrutura seguinte. Transportando isto para a linguagem Python, teríamos então a estrutura `"else-if"`.

Sintaxe. Para que a regra da endentação não seja quebrada, a linguagem usa o termo `"elif"` montado com a sintaxe seguinte:

```
if condição1:
    bloco de comandos 1
elif condição2:
```

```
        bloco de comandos 2
...
elif condição n:
        bloco de comandos n
else:
        bloco de comandos n+1
```

Alguns programas anteriormente apresentados já mostraram este encaixe de estruturas de seleção. Por exemplo, o trecho do programa `L312_04.py` que determina a menor idade está escrito com o formato:

```
...
# Determina a menor idade:
if (A < B) and (A < C): menorIdade = A
else:
    if (B < C): menorIdade = B
    else: menorIdade = C
...
```

O trecho "`else: if`" pode ser reescrito como "`elif`":

```
...
# Determina a menor idade:
if (A < B) and (A < C): menorIdade = A
elif (B < C): menorIdade = B
else: menorIdade = C
...
```

A maior utilidade desse formato é quando há o encadeamento de uma quantidade maior de estruturas de seleção, como o caso mostrado no ***Exemplo 3.6***.

Experimento 01

O programa seguinte implementa o algoritmo dado no ***Exemplo 3.6***. De acordo com a tabela apresentada naquele exemplo, verifica e mostra a categoria de peso de uma pessoa, dados seu peso e sua altura.

Criar o arquivo `L313_01.py` e digitar as seguintes linhas de código:

```
peso = float(input('Digite o peso(kg): '))
altura = float(input('Digite a altura(m): '))
IMC = peso/(altura**2)
if   IMC < 18.5: print ('Categoria: Com baixo peso')
elif IMC < 25.0: print ('Categoria: Com peso normal')
elif IMC < 30.0: print ('Categoria: Com sobrepeso')
elif IMC < 40.0: print ('Categoria: Obeso')
```

```
else: print ('Categoria: Obeso mórbido')
```

Execução de `L313_01.py`. Executando o programa e entrando com os dados seguintes de uma pessoa que tem um peso de 74 kg e altura de 1,70m, visualizamos:

```
>>>
Digite o peso(kg): 74
Digite a altura(m): 1.7
Categoria: Com sobrepeso
>>>
```

Se executarmos este programa seguidamente para duas pessoas, uma com 54 Kg de peso e 1,59m de altura e outra com 110kg de peso e 1,65m de altura, obtemos:

```
>>>
Digite o peso(kg): 54
Digite a altura(m): 1.59
Categoria: Com peso normal
>>>
Digite o peso(kg): 110
Digite a altura(m): 1.65
Categoria: Obeso mórbido
>>>
```

Exercício de autoavaliação

Elabore os programas abaixo com base nos conhecimentos construídos nesta subunidade. Experimente previamente e discuta em seguida no fórum dos conteúdos da semana.

1 - Elabore uma nova versão do programa desenvolvido do item 2 do exercício de autoavaliação da subunidade anterior (o problema do cálculo do peso ideal) admitindo, desta vez, que a informação sobre sexo possa conter erro inserido pelo usuário do programa. Nesse caso, se o sexo for diferente de 1 ou 2, o programa deverá ser emitir a mensagem: “Sexo inválido!”;

2 - Elabore uma nova versão do programa anterior, fazendo apenas uma pequena modificação: A informação sobre o sexo seja "M" ou "m", para masculino, e "F" ou "f", para feminino;

3 - Numa certa faculdade, o conceito de um estudante é calculado a partir da nota numérica, de acordo com a seguinte tabela:

Nota	Conceito
Maior ou igual a 8,5	A
Menor que 8,5, porém maior ou igual a 7,0	B
Menor que 7,0, porém maior ou igual a 5,0	C
Menor que 5,0, porém maior ou igual a 3,5	D

Menor que 3,5	E

Elabore um programa em Python para ler a nota de um aluno e, usando esta tabela, escrever o conceito correspondente. Obs.: Se for digitada uma nota fora da faixa de 0.0 a 10.0, o programa deverá emitir a mensagem "`Nota inválida`".

Unidade III.2

Estruturas de repetição

Apenas reforçando algo que já mencionamos no início do curso, uma estrutura de repetição provoca a repetição de comandos, respeitando uma dada condição. Em outras palavras, essa estrutura executa várias vezes um mesmo bloco de comandos, sendo a quantidade de execuções determinada pela obediência à condição. Os dois casos de repetição seguintes caracterizam, respectivamente, dois tipos de estruturas de repetição:

O primeiro é quando as execuções do bloco de comandos acontecem "para" um conjunto conhecido de valores de uma dada variável. Isto é, a condição está amarrada diretamente à quantidade de repetições. Chamaremos sugestivamente esta estrutura de estrutura *para*.

O outro caso é quando as execuções do bloco de comandos acontecem "enquanto" a condição permanecer respeitada (verdadeira). Somente um fato novo dentro do bloco de comandos tem o poder de fazer a condição ser contrariada (se tornar falsa) e somente assim as repetições serem encerradas. Isto é, a condição de parada da estrutura não está diretamente ligada à quantidade de repetições (por isso não se sabe de antemão a quantidade de repetições). Chamaremos esta estrutura de estrutura *enquanto*.

3.2.1 A estrutura "para"

Normalmente, usamos esta estrutura quando, claramente, o problema permite a antevisão da quantidade necessária de repetições de um certo conjunto de ações que levará à sua solução.

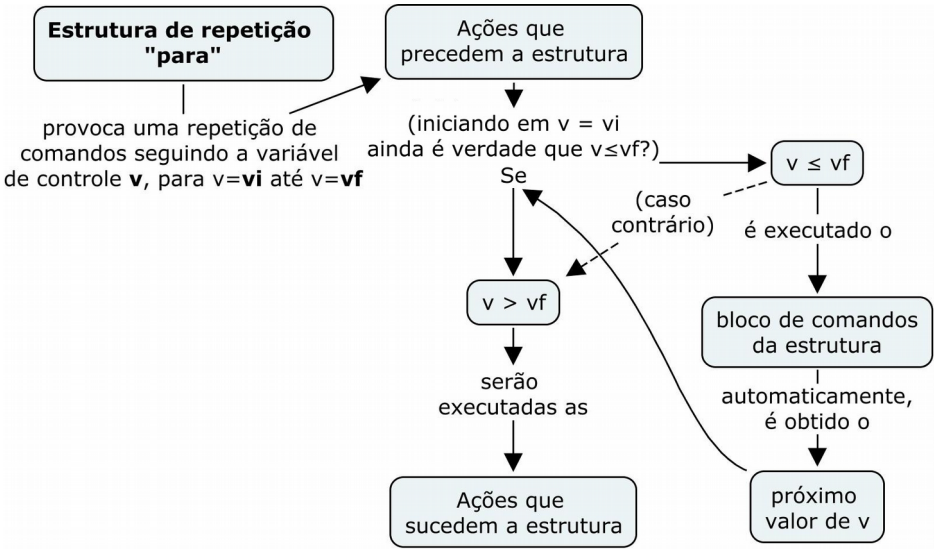
Vejamos alguns desses casos. Por exemplo, se quisermos:

-*Tirar o suco de 10 laranjas*: A solução passa por repetir o ato de espremer uma laranja, em que isto é feito *para* a primeira laranja, a segunda, até a décima (são 10 execuções);

-*Escrever os cinco primeiros números primos*: A solução passa por escolher um número primo e escrevê-lo. Em ordem crescente, repetimos estas ações *para* o primeiro, o segundo, etc, até o quinto número, quando paramos;

-*Calcular a média de consumo em KWh de uma amostra de 200 consumidores de energia elétrica*: A solução passa por acumular em um somatório os KWh's de um consumidor. Esta ação é então repetida *para* o consumidor de número 1, 2, etc, até o de número 200. Ao final da repetição, dividimos a soma de todos os consumos em KWh pela quantidade de consumidores;

Percebemos nitidamente que a propriedade comum desses problemas é que a quantidade necessária de repetições é conhecida desde o início. Do ponto de vista computacional, portanto, precisamos de um "contador" de execuções e, para esta função, normalmente usamos uma variável, chamada de *variável de controle*. Na verdade, a contagem consiste em definir, como condição da estrutura, que a variável de controle deve pertencer obrigatoriamente a um conjunto de valores declarados previamente. Qualquer um dos problemas citados acima pode se encaixar na estrutura descrita no mapa seguinte:



Sintaxe

Nos algoritmos, a estrutura **"para"** pode ser descrita com a seguinte sintaxe:

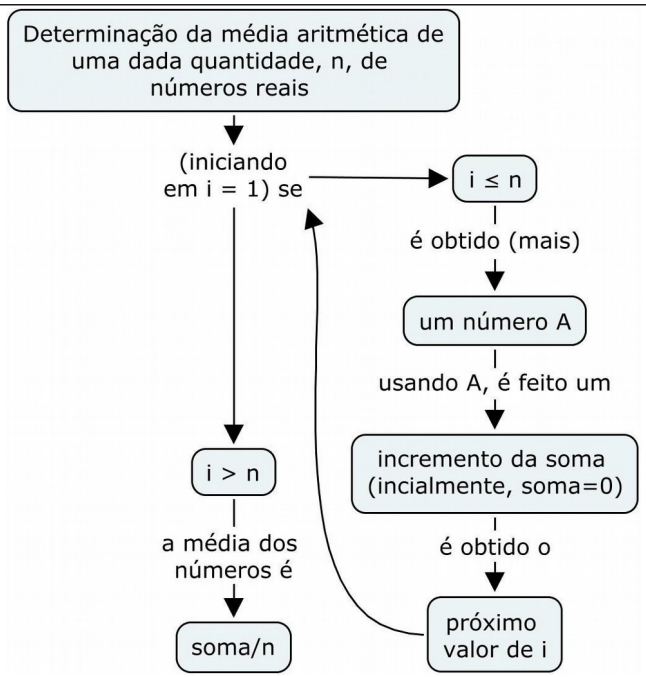
```
para v ← vi...vf, faça:
    bloco de comandos
```

Onde: A condição amarrada à estrutura é que cada valor assumido pela variável de controle **v** deve pertencer ao conjunto de valores em que v_i é o valor inicial e v_f é o valor final. O **bloco de comandos** será executado para cada valor da variável de controle **v**.

Consideremos o exemplo seguinte, onde calculamos a média aritmética de um conjunto de números reais.

Exemplo 3.7

Desejamos calcular a média aritmética de uma dada quantidade de números reais. Para tanto, precisamos calcular a soma desses números. A média aritmética será obtida dividindo-se esta soma pela quantidade de números (que é conhecida). A solução desse problema está descrita no mapa seguinte:



A solução algorítmica pode ser a seguinte:

```
Algoritmo
1 escrever "Digite a quantidade de números"
2 ler n
3 soma ← 0.0
4 escrever "Digite os números"
5 para i ← 1...n, faça:
      ler A
      soma ← soma + A
6 escrever "A média dos números é", soma/n
Fim-Algoritmo
```

O primeiro comando do algoritmo solicita ao usuário que ele digite qual é quantidade `n` de números desejada. O segundo comando faz a leitura do valor. O terceiro passo prepara a variável `soma` para conter o somatório dos números a serem lidos, iniciando a mesma com o valor 0.0 (zero). Vale observar que, em computação, é aceito o termo *inicialização* para designar atribuição de valores iniciais, como é o caso da variável `soma` (isto é, `soma` foi *inicializada* com 0.0). O quarto passo solicita do usuário que digite os números a serem inicialmente somados (as ações de leitura e soma dos números só vão acontecer no passo seguinte).

Vamos supor que desejamos calcular a média aritmética de apenas três números reais. Após a execução desses primeiros passos, as variáveis `n` e `soma` estarão preenchidas da seguinte maneira (as variáveis `i` e `A` ainda não existem na memória):

N	i	A	soma
3	-	-	0.0

O quinto passo é uma estrutura de repetição. `i` é a variável de controle. Para cada valor assumido pela variável `i` (que são os números inteiros de `1` até o valor `n`), a estrutura executará um bloco com apenas dois comandos. O primeiro comando do bloco é a leitura de um dos números (chamado de `A`) e o segundo comando é a inserção desse número no somatório `soma`. O conteúdo de `soma` é modificado da maneira como vimos no ***Exemplo 2.16***, do **Módulo II** (seu novo valor é resultado da soma de `A` com seu valor antigo) e, por isso, foi necessária a atribuição de um valor inicial a esta variável. Por exemplo, se quisermos calcular a média dos números `325.1`, `18.3` e `128.0`, devemos digitá-los seguidamente e o quadro de variáveis se comportará da seguinte maneira:

Passos do algoritmo:	Var. **n**	Var. **i**	Var. **A**	Var. **Soma**	*Observações:*
1º. passo	-	-	-	-	Escreve a mensagem: "Digite a quantidade..."
2º. passo	**3**	-	-	-	Leitura de `n` (no caso, `n` corresponde a três números)
3º. passo		-	-	`0.0`	Inicialização de `soma`
4º. passo		-	-		Escreve a mensagem: "Digite os números."
5º. passo, 1ª. execução		`1`	`325.1`	`325.1`	`Soma = 0.0 + 325.1 = 325.1`
5º. passo, 2ª. execução		`2`	`18.3`	`343.4`	`Soma = 325.1+ 18.3 = 343.4`
5º. passo, 3ª. execução		`3`	`128.0`	**`471.4`**	`Soma = 343.4+128.0 = 471.4`
6º. passo		`4`			Escreve o resultado: "A média dos números..."
Estado final da memória	**3**	`4`	`128`	**`471.4`**	

No quadro acima, o sexto passo é a escrita do resultado. Vemos que o último valor assumido pela variável `soma` é `471.4`. Logo, o resultado no caso daqueles números é:

```
A média dos números é 157.13333
```

(pois `471.4/3` é aproximadamente igual a 157.13333).

O algoritmo acima só não está protegido contra divisão por zero. Se, por engano, o usuário digitar o zero para o valor de `n`, nem haverá números para serem digitados. Também não queremos alterar o sinal do resultado. Por isso, o resultado somente deve ser exibido se `n` for positivo (nem negativo, nem zero). Nesse caso, podemos reescrever a solução do problema conforme o exemplo seguinte.

Exemplo 3.8

O algoritmo abaixo é uma nova versão do algoritmo dado no ***Exemplo 3.7***. Desta vez, o algoritmo somente prossegue com os demais comandos se a quantidade **`n`** for um número positivo.

```
Algoritmo
```

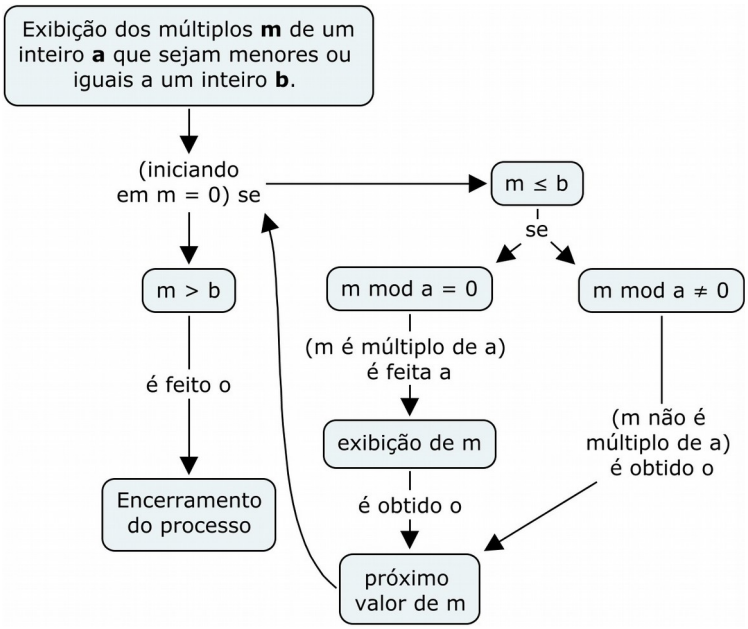
```
1 escrever "Digite a quantidade de números"
2 ler n
3 se (n > 0) então:
3.1   soma ← 0.0
3.2   escrever "Digite números"
3.3   para i ← 1...n, faça:
            ler A
            soma ← soma + A
3.4   escrever "A média dos números é", soma/n
Fim-Algoritmo
```

No exemplo acima, os comandos 3.1 a 3.4 (que são correspondentes aos passos 3 a 6 do algoritmo do ***Exemplo 3.7***) somente serão executados se o valor de `n` for positivo.

Exemplo 3.9

Consideremos o seguinte problema: Dados dois números inteiros positivos, `a` e `b`, escrever todos os múltiplos de `a` que sejam menores ou iguais a `b`.

De antemão, sabemos que os múltiplos do número `a` são todos os números divisíveis por `a` (excluímos `a` = 0) que, por definição, são números naturais (correspondem aos inteiros que não são negativos). Por exemplo, os múltiplos de 3 são 0, 3, 6, etc. (confere que todos são divisíveis por 3, ou, a divisão de qualquer um deles por 3 é exata). O detalhe do problema é que se pede apenas os primeiros múltiplos e o maior valor não ultrapassa `b`. Logo, a solução é percorrer todos os números naturais na faixa de `0` a `b` e escrever cada número `m` que tornar verdadeira a expressão: `m mod a = 0` (ou seja, cada número cuja divisão por `a` produzir resto igual a zero), como mostra o mapa seguinte:



Podemos então expressar esta solução através do seguinte algoritmo:

```
Algoritmo
1 escrever "Digite dois números inteiros positivos:"
2 escrever "a = "
3 ler a
4 escrever "b = "
5 ler b
6 escrever "Os múltiplos de",a,"menores ou iguais a",b,"são:"
7 para m ← 0...b, faça:
      se (m mod a = 0) então:
            escrever m
Fim-Algoritmo
```

Usando o algoritmo acima para escrever, por exemplo, os múltiplos de 3 menores ou iguais a 4, teríamos a seguinte situação:

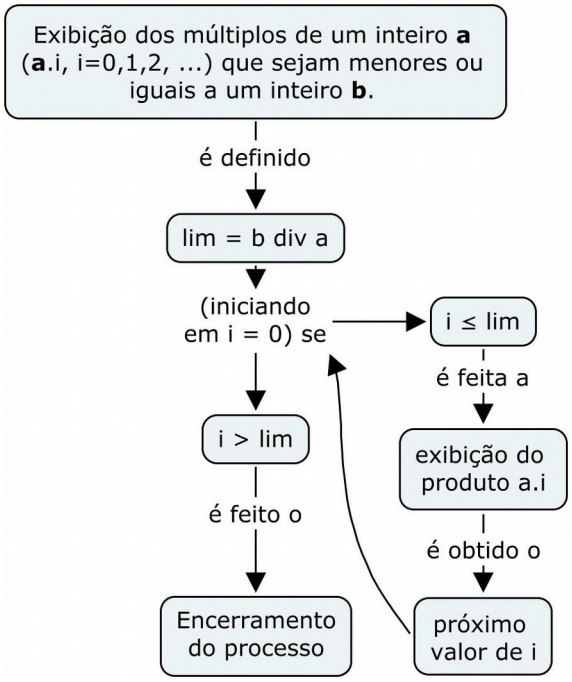
Passos do algoritmo:	Var. **a**	Var. **B**	Var. **m**	É verdade que **m** mod **a = 0**?	Observações:
1º. ao 6º.passo	3	4	-	-	Emissões de mensagens, leitura de a e leitura de b
7º. passo, 1ª. execução			0	**0** mod 3 = 0 (é verdade)	Escreve **0**
7º. passo, 2ª. execução			1	1 mod 3 = 1	Descarta 1
7º. passo, 3ª. execução			2	2 mod 3 = 2	Descarta 2
7º. passo, 4ª. execução			3	**3** mod 3 = 0 (é verdade)	Escreve **3**
7º. passo, 5ª. execução			4	4 mod 3 = 1	Descarta 4

As soluções algorítmicas normalmente não são únicas. Podemos resolver o mesmo problema do exemplo acima aplicando um outro algoritmo. Vejamos porque. Para resolver o problema do ***Exemplo 3.9*** nos baseamos na definição de que os múltiplos de `a` são os números divisíveis por a. Isto é equivalente a dizer que os múltiplos de `a` são os números que resultam do produto de `a` por um número natural. Por exemplo, os múltiplos de 3 são 0 (ou, 3x0), 3 (ou, 3x1), 6 (ou, 3x2), etc. Logo, para obter os múltiplos podemos nos guiar pelos fatores 0, 1, 2, 3, etc., faltando apenas encontrar um limite, `lim`, para esta sequência. Limitamos a sequência usando o valor `b` da seguinte maneira: `lim` é o número que, multiplicado por `a`, resulta num valor menor ou igual a `b`. Assim, os múltiplos de `a` serão os números `a*i`, com `i` variando de `0` até `lim`.

Exemplo 3.10

Vamos resolver o mesmo problema do exemplo anterior usando desta vez a sequência de fatores que produzem os múltiplos de `a`. Observemos que o limite da sequência é calculado pela expressão `b div a`. Ou seja, `lim` é o quociente da divisão inteira de `b` por `a` (verificamos

que `a*lim` é menor ou igual a `b`, uma solução inspirada no resultado do ***Exemplo 2.15***). Vejamos um mapa que demonstra esta solução e, em seguida, o algoritmo correspondente.




```
Algoritmo
1 escrever "Digite dois números inteiros positivos:"
2 escrever "a = "
3 ler a
4 escrever "b = "
5 ler b
6 lim ← b div a
7 escrever "Os múltiplos de",a,"menores ou iguais a",b,"são:"
8 para i ← 0...lim, faça:
      escrever a*i
Fim-Algoritmo
```

Aplicando agora a versão acima, que usa o cálculo dos fatores, para determinar os múltiplos de 3 (valor de `a`) menores ou iguais a 4 (valor de `b`), teremos:

Passos do algoritmo:	Var. **a**	Var. **B**	Var. **lim**	Var. **i**	**a*i**	**Observações:**
1º. ao 5º. passo	3	4	-	-	-	Emissões de mensagens, leitura de `a` e leitura de `b`
6º. ao 7º. passo			1	-	-	Cálculo de `lim` e emissão de mensagem
8º. passo, 1ª. execução				0	3*0 = **0**	Escreve **0**
8º. passo, 2ª. execução				1	3*1 = **3**	Escreve **3**

Os algoritmos nas duas versões acima ainda podem ser incrementados. Por exemplo, podemos obrigar que o valor `a` seja diferente de zero. Com isso, atenderíamos a um problema de definição matemática e também à restrição para cálculo das expressões (`m mod a` no

Exemplo 3.9 e `b div a` no ***Exemplo 3.10***) que provocariam divisões por zero. Isto seria uma proteção contra enganos de digitação do valor de `a`, como no exemplo seguinte.

Exemplo 3.11

O algoritmo abaixo é o resultado de uma melhoria aplicada ao ***Exemplo 3.10***. Desta vez, após a leitura do primeiro dado de entrada, os demais comandos somente serão executados se o valor de a for um inteiro positivo. O algoritmo também aproveita para emitir uma mensagem no caso de o valor digitado ser negativo

```
Algoritmo
1 escrever "Digite dois números inteiros positivos:"
2 escrever "a = "
3 ler a
4 se (a > 0) então:
4.1   escrever "b = "
4.2   ler b
4.3   lim ← b div a
4.4   escrever "Os múltiplos de",a,"menores ou iguais a",b,"são:"
4.5   para i ← 0...lim, faça:
            escrever a*i
  senão:
4.6   escrever "Dado inválido!"
Fim-Algoritmo
```

Laboratório - Estrutura de repetição "para"

Objetivos

Conferir o uso da estrutura de repetição “para”, dado que a condição de interrupção das iterações está baseada no conhecimento da quantidade destas.

Usar os recursos disponíveis na linguagem Python para aplicar os conceitos e testar os algoritmos desta subunidade.

Recursos e experimentação

Vimos que a estrutura de repetição “para” requer a prévia definição de um intervalo de valores para a sua variável de controle. Isto é, a variável de controle **`v`** deve assumir seguidamente todos valores dentro da faixa $\mathbf{v_i...v_f}$, onde $\mathbf{v_i}$ é o valor inicial e $\mathbf{v_f}$ é o final.

Em Python, uma das maneiras de prover os valores $\mathbf{v_i...v_f}$, é através de uma listagem direta, usando o tipo estruturado denominado de ***`list`***. Avançaremos um pouco mais sobre as listas de Python no próximo módulo desta disciplina. Por enquanto, podemos adiantar que o tipo `list` é definido utilizando-se colchetes e admite os mais diversos tipos de dados

numa mesma lista. Outro fato importante é podermos verificar se um determinado elemento pertence ou não a uma lista usando o operador lógico **in**.

São exemplos de listas: `['Outubro', 'Novembro', 'Dezembro']`, `[3, 45, 56, 18, -5]` e `[2009, '@', 'Alagoas', 45]`. Usando o operador **in**, podemos estabelecer relações como (`'@' in [2009, '@', 'Alagoas', 45]`) e (`'Janeiro' in ['Outubro', 'Novembro', 'Dezembro']`). Obs.: Os valores lógicos dessas relações são, respectivamente, `True` (verdadeiro, ou seja, pertence) e `False` (falso, ou seja, não pertence).

Realizemos então o experimento seguinte objetivando melhor apreciar estas propriedades das listas.

Experimento 01

Usemos o interpretador Python interativamente para experimentar algumas listas:

```
>>> print ([2009, '@', 'Alagoas', 45]) # Apenas escreve a lista
[2009, '@', 'Alagoas', 45]
>>> '@' in [2009, '@', 'Alagoas', 45]      # '@' está na lista?
True
>>> 'Janeiro' in ['Outubro', 'Novembro', 'Dezembro']
False
>>> vogais = ['a', 'e', 'i', 'o', 'u']     # vogais é uma lista
>>> print (vogais)              # mostra a lista chamada vogais
['a', 'e', 'i', 'o', 'u']
>>> 'b' in vogais                                  # 'b' é vogal?
False
>>>
```

Claramente, podemos notar que a listagem direta da faixa de valores é interessante para enumeração de elementos de tipos diversos e em pequena quantidade (se tivermos centenas ou milhares de elementos, por exemplo, isto pode se tornar inviável, ou talvez seja necessário o exame caso a caso). Considerando que é desejável a construção de sequências de tamanhos quaisquer e que o caso numérico é o mais frequente, a linguagem Python disponibiliza a construção de *progressões aritméticas, P.*A. (consulte a b*ibliografia complementar*), através da função

`range(a, b, passo)`,

onde:

- **`a`** e **`b`** definem um intervalo de valores inteiros. Os elementos são os inteiros *maiores ou iguais* a **`a`** e *menores* que **`b`**;
- O parâmetro **`passo`** se refere ao incremento do valor inicial **`a`**.

Variações: Omitindo-se o parâmetro **a**, assume-se **a** = 0. Omitindo-se o parâmetro **passo** assume-se **passo** = 1. Se **passo** for diferente de 1 e **a** for zero, o valor **a** = 0 não pode ser omitido.

Continuemos nosso experimento. Devemos conhecer os efeitos da função `range()`. Convém observar que ela define uma regra para obtenção de uma P.A.. O propósito das execuções abaixo é tornar isso visível. Para tanto, quando precisarmos explicitar a saída da função, faremos uma conversão desta com `list()`:

```
>>> list(range(1,11)) # Inteiros a partir de 1 e menores que 11
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(1,11, 3))            # Agora, contados de 3 em 3.
[1, 4, 7, 10]
>>> list(range(4, -4, -1))       # Sequência com passo negativo.
[4, 3, 2, 1, 0, -1, -2, -3]
>>> list(range(10))                 # Lista de inteiros de 0 a 9
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> 7 in range(1,11, 3)       # 7 pertence à lista [1, 4, 7, 10]
True
>>> 10 in range(10) # 10 (limite sup.) não pertence ao intervalo
False
```

Revendo um pouco, notamos que, para exibir o efeito da função `range()`, convertemos sua saída em lista. No entanto, nas relações de pertinência, esta conversão não é necessária.

Agora sim, de posse desse prévio conhecimento sobre listas, podemos expor sobre a sintaxe da estrutura “para” no caso da linguagem Python.

Sintaxe. A linguagem Python implementa a estrutura de repetição “para” sob o formato da estrutura “for” seguinte:

```
for v in lista_de_valores:
    bloco de comandos
```

Onde: **v** é a variável de controle, que assume sequencialmente os valores pertencentes a ***lista_de_valores***. O **bloco de comandos** será executado para cada valor de **v**.

No experimento seguinte, teremos a oportunidade de implementar o algoritmo do ***Exemplo 3.7***.

Experimento 02

O programa abaixo calcula e exibe a média aritmética de `n` números reais. A leitura e a soma de cada número lido são os comandos da estrutura de repetição, e serão repetidos `n`

vezes. No ***Exemplo 3.7*** essas `n` vezes é conseguida fazendo-se a variável `i` assumir os valores inteiros 1, 2, 3, ... até `n`.

Na implementação, querendo seguir essa mesma sequência, `[1, 2, 3,..., n]`, é bastante usar a chamada `range(1, n+1)`. No entanto, para a solução desse problema, o relevante é a quantidade `n` de iterações, não se fazendo uso direto dos valores assumidos pela variável de controle `i`. Por isso, a definição de uma lista qualquer com `n` elementos é suficiente. Então, a chamada `range(n)` é a solução adequada porque produz a lista `[0, 1, 2,..., n-1]` com, exatamente, `n` elementos.

Criar o arquivo `L321_02.py` e digitar as seguintes linhas de código:

```
n = int(input('Digite a quantidade de números '))
soma = 0.0
print ('Digite os números')
for i in range(n):
    A = float(input())
    soma += A
print ('A média dos números é', soma/n)
```

Execução de `L321_02.py`. Executando o programa, calcularemos a média aritmética dos números `325.1`, `18.3` e `128.0`, já rastreados no ***Exemplo 3.7***. Digitando esses valores, obtemos o resultado abaixo:

```
>>>
Digite a quantidade de números 3
Digite os números
325.1
18.3
128.0
A média dos números é 157.133333333
>>>
```

A única diferença da solução acima em relação ao ***Exemplo 3.8*** é que temos neste último um teste verificando de o usuário digitou o valor `n` corretamente. A implementação, portanto, pode ser deduzida do programa `L321_02.py`. Veremos a seguir a implementação do ***Exemplo 3.9***.

Experimento 03

O programa abaixo escreve todos os múltiplos de um número `a` que são menores ou iguais a um dado número `b`. Sabendo-se que o menor múltiplo de qualquer inteiro é o zero, a solução consiste em percorrer todos os inteiros de `0` até `b` e selecionar, e escrever, somente aqueles que forem divisíveis por `a`. Para esta finalidade, a função `range()` com um parâmetro é adequada, faltando apenas fazer um ajuste devido ao fato de que o valor `b` tem que pertencer

à lista. Explicando melhor, se usássemos `range(b)`, de acordo com sua definição na linguagem, o intervalo de valores seria `[0, 1, 2, ..., b-1]`. Assim, para produzir a sequência `[0, 1, 2, ..., b]`, o uso da função range deve ser: `range(b+1)`.

Criar o arquivo `L321_03.py` e digitar as seguintes linhas de código:

```
print ('Digite dois números inteiros positivos:')
a = int(input('a = '))
b = int(input('b = '))
print ('O múltiplos de',a,'menores ou iguais a',b,'são:')
for m in range(b+1):
    if m % a == 0: print(m)
```

Execução de `L321_03.py`. Executando o programa com os valores testados no ***Exemplo 3.9*** (a = 3 e b = 4), visualizamos o resultado abaixo (confere com o quadro mostrado no referido exemplo):

```
>>>
Digite dois números inteiros positivos:
a = 3
b = 4
O múltiplos de 3 menores ou iguais a 4 são:
0
3
>>>
```

Experimento 04

O programa abaixo implementa o algoritmo do ***Exemplo 3.10***. Os múltiplos de um número a são obtidos a partir da sequência de fatores cujo limite é lim. Isto é, os múltiplos de a são: a*0, a*1, a*2, ..., a*lim. lim é obtido a partir da divisão inteira b//a. Considerando que o valor máximo deve ser incluído na lista de valores [0, 1, 2, ..., lim], o uso adequado da função `range()` será: `range(lim+1)`.

Criar o arquivo `L321_04.py` e digitar as seguintes linhas de código:

```
print ('Digite dois números inteiros positivos:')
a = int(input('a = '))
b = int(input('b = '))
lim = b//a
print ('Os múltiplos de',a,'menores ou iguais a',b,'são:')
for i in range(lim+1): print (a*i)
```

Execução de `L321_04.py`. Executando o programa com os valores testados no ***Exemplo 3.10*** (a = 3 e b = 4), visualizamos o resultado abaixo (confere com o quadro mostrado no referido exemplo):

```
>>>
Digite dois números inteiros positivos:
a = 3
b = 4
Os múltiplos de 3 menores ou iguais a 4 são:
0
3
>>>
```

Experimento 05

Uma versão mais completa do programa acima pode ser obtida com a implementação do ***Exemplo 3.11***.

Criar o arquivo `L321_05.py` e digitar as seguintes linhas de código:

```
print ('Digite dois números inteiros positivos:')
a = int(input('a = '))
if a > 0:
    b = int(input('b = '))
    lim = b//a
    print ('O múltiplos de',a,'menores ou iguais a',b,'são:')
    for i in range(lim+1): print (a*i)
else: print ('Dado inválido!')
```

Execução de `L321_05.py`. Executando o programa e entrando com valores que não sejam positivos, o programa emitirá uma mensagem. Por exemplo, digitando `a = 0`:

```
>>>
Digite dois números inteiros positivos:
a = 0
Dado inválido!
>>>
```

Exercício de autoavaliação

Com base nos conhecimentos construídos nesta subunidade, elabore programas que resolvem os problemas abaixo. Experimente previamente e discuta em seguida no fórum dos conteúdos da semana.

1 - Dados três números inteiros e positivos, `a`, `b` e `c`, escrever todos os múltiplos de `a` que sejam maiores ou iguais `b` e menores ou iguais a `c`.

2 - A partir de uma amostra de 24 meses de consumo de energia elétrica (em KWh) de um determinado usuário, calcular a média e o maior consumo registrado (*Sugestão quanto ao procedimento de escolha do maior consumo*: *Cada consumo lido do teclado é candidato a ser*

o maior consumo. Logo, deve-se designar uma variável nova para registrá-lo, iniciando a mesma com zero – que seria o consumo mínimo).

3.2.2 A estrutura "enquanto"

Esta estrutura é aplicável nos casos em que o problema não permite prever a quantidade necessária de repetições dos comandos que levam à solução. A condição de parada é totalmente desvinculada da quantidade de repetições.

Tomemos o ***Exemplo 1.1*** do telefonema (da **Subunidade I.2.2**). Segundo aquelas condições, uma pessoa que deseja se comunicar por telefone com outra irá repetir a discagem quantas vezes for necessário. A repetição da discagem irá acontecer enquanto o contato não for estabelecido (ou seja, enquanto a chamada não for atendida, ou for atendida, mas não pela pessoa desejada).

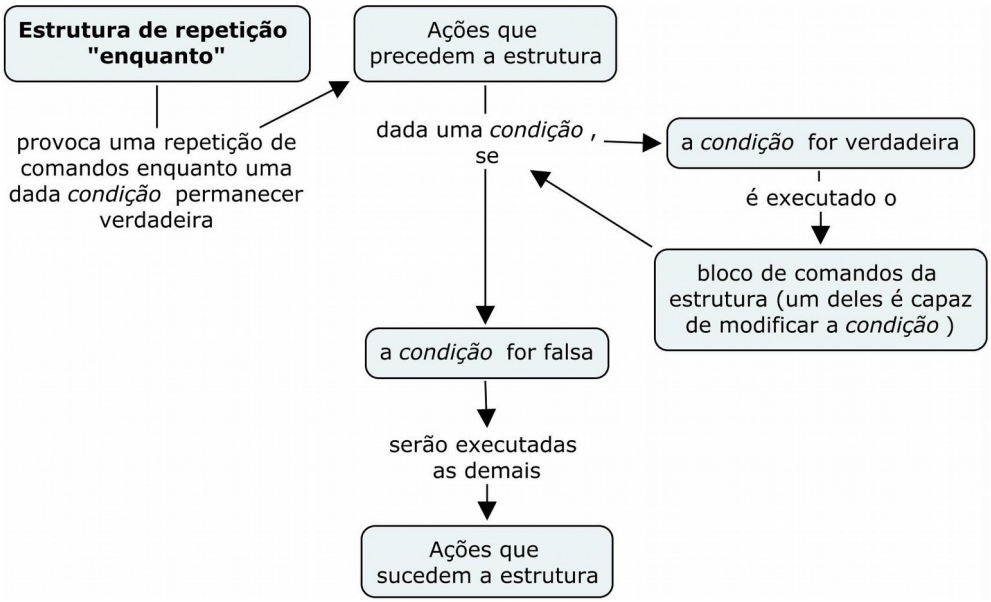
Se quisermos, por exemplo:

-*Tirar o suco de laranjas até completar um litro de suco (sabendo-se que há laranjas suficientes para tanto)*: A condição de parada está desvinculada da quantidade de laranjas. Não se sabe quantas laranjas irão completar um litro de suco (não se conhece exatamente qual é o volume de suco de uma laranja). A solução passa por repetir o ato de espremer uma laranja. Espremem os a primeira, a segunda, etc, *enquanto* não completar um litro de suco (e paramos assim que o objetivo for alcançado).

- *Calcular a média de consumo em KWh de uma quantidade indeterminada consumidores de energia elétrica, sabendo-se apenas que o consumo será definido como nulo (0 KWh), quando não houver mais KWh's a contabilizar*: A condição de parada está desvinculada da quantidade de consumidores. Não se sabe previamente quantos consumidores terão seus consumos contabilizados quando a lista acabar. Portanto, enquanto se lê seus consumos, os consumidores precisam ser contados (pois queremos calcular uma média). Então, a solução passa por acumular os KWh's de cada consumidor em um somatório e a contagem de consumidores em outro. Estas somas serão efetuadas com referência ao consumidor de número 1, 2, etc., *enquanto* não for informado o fim da lista (que será indicado com o consumo zero). Parando a repetição, dividimos a soma dos consumos pela quantidade de consumidores;

Nesses problemas, a quantidade necessária de repetições é desconhecida desde o início, todavia é importante lembrar que sempre há um limite para o número repetições (caso contrário, a solução algorítmica não seria impossível!). O fato de não termos conhecimento prévio do número de repetições deve então ser compensado por um outro critério que permita monitorar cada repetição e indicar o fim das iterações. Ficamos livres até para usar um contador de iterações quando o problema exigir. Este é o caso do cálculo da média de consumo de energia, citado logo acima. Precisava-se contar os consumidores.

O mapa seguinte descreve a estrutura "**enquanto**":



Sintaxe

Nos algoritmos, a estrutura acima citada pode ser descrita com a seguinte sintaxe:

```
enquanto (condição) faça:
    bloco de comandos
```

Onde: A **condição** amarrada à estrutura é dada por um valor lógico (proveniente de uma expressão lógica, uma constante ou uma variável do tipo lógico). As repetições acontecerão enquanto a condição estiver sendo atendida (ou seja, com valor lógico `verdadeiro`).

O **bloco de comandos** será executado repetidamente enquanto a **condição** da estrutura permanecer com valor `verdadeiro` (a repetição é interrompida quando o valor da condição se tornar `falso`).

É importante notar o seguinte. Uma vez passado o controle do algoritmo para essa estrutura, os comandos do bloco serão os únicos a serem executados. Isto significa também que a única chance de a `condição` mudar de valor é através de um desses comandos. Não existindo um comando com esta incumbência, a `condição` nunca mudará de valor e a estrutura entrará no que chamamos de *loop* infinito.

Vamos considerar o seguinte problema, cuja tarefa de resolução pode ajudar a resolver outros que envolvem uma lógica semelhante: calcular a média aritmética de uma quantidade indeterminada de números reais. O objetivo é o mesmo do ***Exemplo 3.6*** (que é calcular a média aritmética), mas, desta vez, a quantidade de números não é dada inicialmente. Não havendo conhecimento prévio da quantidade, enquanto se lê os números, estes precisam ainda ser contados. Logo, é necessário preparar dois "acumuladores": um, para a soma dos próprios números e outro para a contagem destes. O resultado que se espera será obtido dividindo o somatório dos números pelo somatório que representa a quantidade destes. Falta apenas decidir como avisar o fim da leitura dos números!

Geralmente, quando o assunto é a leitura de dados, podemos avisar que não temos mais dados a inserir usando uma *interrupção interna* aos dados (ou, técnica do *flag*) ou uma *interrupção externa* aos dados:

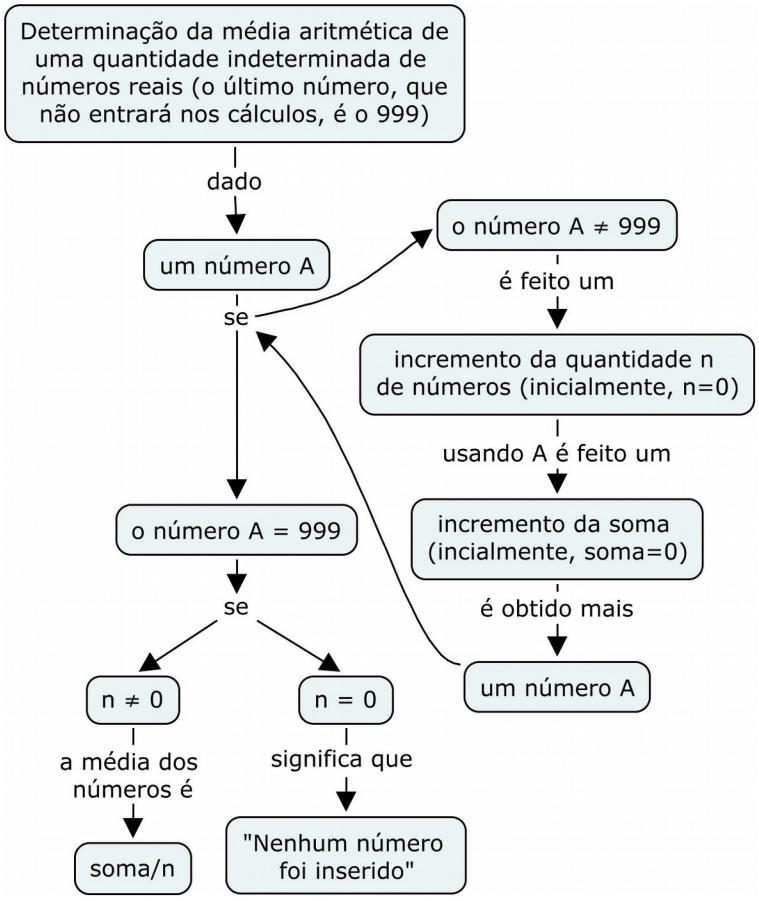
Interrupção interna aos dados. Usando esta técnica, colocamos uma marca nos próprios dados. Isto é, escolhemos um deles (o *flag*) para que sirva apenas de indicador do fim da leitura. No exemplo acima, um dos números deverá servir de *flag* e, logicamente, este não entrará nos cálculos. O *flag* é muito usado em programação de computadores porque é uma das maneiras mais fáceis de aviso sobre uma determinada característica na captura, armazenamento ou processamento de dados.

O exemplo seguinte apresenta o problema do cálculo da média aritmética de um conjunto de números, mostrando um caso típico de interrupção das iterações por uma marca nos dados.

Exemplo 3.12

Retomemos o problema do cálculo da média aritmética com o seguinte enunciado: Calcular a média aritmética de uma quantidade indeterminada de números reais, onde o último número, que não entrará nos cálculos, é o 999.

A solução desse problema pode ser descrita no mapa seguinte:



A indicação do fim das iterações está nos próprios dados, como também está exibido no algoritmo abaixo. Usamos o *flag* 999 da seguinte maneira: Lemos o primeiro número e, se

não for o *flag*, prosseguiremos com a leitura. Continuando, os demais números somente deverão ser acumulados e contados *enquanto* não for o *flag*:

```
Algoritmo
1 n ← 0
2 soma ← 0.0
3 escrever "Digite os números (999 - encerra):"
4 ler A
5 enquanto (A ≠ 999) faça:
5.1    n ← n + 1
5.2    soma ← soma + A
5.3    ler A
6 se (n ≠ 0) então:
       escrever "A média dos números é", soma/n
  senão:
       escrever "Nenhum número foi digitado"
Fim-Algoritmo
```

Nos passos 1 e 2 do algoritmo acima, as variáveis `n` e `soma` são criadas (`n` contará os números e `soma` acumulará seus valores), ambas com valor nulo (ainda não foi lido nenhum número nem sabemos quantos serão digitados). O passo 3 solicita digitação dos números avisando que se digitar o número 999 a entrada de dados se encerrará. No passo 4, o primeiro número é lido.

Vemos também que o passo 5 é uma estrutura de repetição que executará os comandos 5.1, 5.2 e 5.3 enquanto cada um dos números digitados *não* for o *flag* 999. Conforme já foi comentado anteriormente, os comandos da estrutura precisam ser fiscalizados no sentido de que um deles tem que abrigar os recursos para modificar o estado da condição de *verdadeiro* para *falso*. A condição da estrutura (`A ≠ 999`, inicialmente verdadeira) envolve o valor de um dos números. Portanto, é o comando 5.3 que oferece a chance modificá-la. Ou seja, o próximo número lido, `A`, pode ser o valor 999 que produzirá a expressão `999 ≠ 999`, falsa, e assim encerrando a repetição.

Para que o controle do algoritmo chegue a entrar na estrutura, no início da iteração, `A ≠ 999` tem que ser verdadeira. Isto só não acontecerá se, excepcionalmente, o primeiro número digitado for o *flag*. Se isto acontecer, nenhum número será somado e a variável `n` se manterá igual a zero. Por isso, no passo 6, um teste foi acrescentado. A média somente é calculada e escrita se `n` não for o zero.

Calculemos, por exemplo, a média dos números `325.1`, `18.3` e `128.0` seguindo o quadro abaixo conforme o algoritmo dado:

Passos do algoritmo:	É verdade que	Var. **n**	Var. **soma**	Var. **A**	Observações:

	A ≠ 999?				
1º. e 2º. passo	–	0	0.0	–	Inicializações de n e soma
3º. e 4º. passo	–	0	0.0	325.1	Emissão de mensagem e leitura de A
5º. passo, 1ª. execução	*325.1* ≠ *999* (é verdade). Ocorrerá uma repetição.	1	325.1	18.3	(5.1) n = 0 + 1 = 1 (5.2) soma = 0.0 + 325.1= 325.1 (5.3) leitura de A
5º. passo, 2ª. execução	*18.3* ≠ *999* (é verdade). Ocorrerá mais uma repetição.	2	343.4	128.0	(5.1) n = 1 + 1 = 2 (5.2) soma = 325.1+18.3 = 343.4 (5.3) leitura de A
5º. passo, 3ª. execução	*128.0* ≠ *999* (é verdade). Ocorrerá mais uma repetição.	**3**	**471.4**	999	(5.1) n = 2 + 1 = 3 (5.2) soma = 343.4+128.0= 471.4 (5.3) leitura de A (lê o *flag*)
–	*999* ≠ *999* (é falso). Não mais ocorrerá repetição.	–	–	–	–
Estado final da memória		**3**	**471.4**	999	

Para encerrar a entrada de dados, foi digitado o valor 999 (terceira execução do passo 5) e assim não houve mais nenhuma repetição (porque A ≠ 999 se tornou falsa). De posse dos valores finais de n (igual a 3) e soma (igual a 471.4) o valor da média aritmética é calculado e a saída é:

```
A média dos números é 157.13333
```

Resolveremos a seguir este mesmo problema usando uma interrupção externa aos dados. O objetivo é comparar esta maneira de resolução com o caso anterior em que foi utilizado um *flag*.

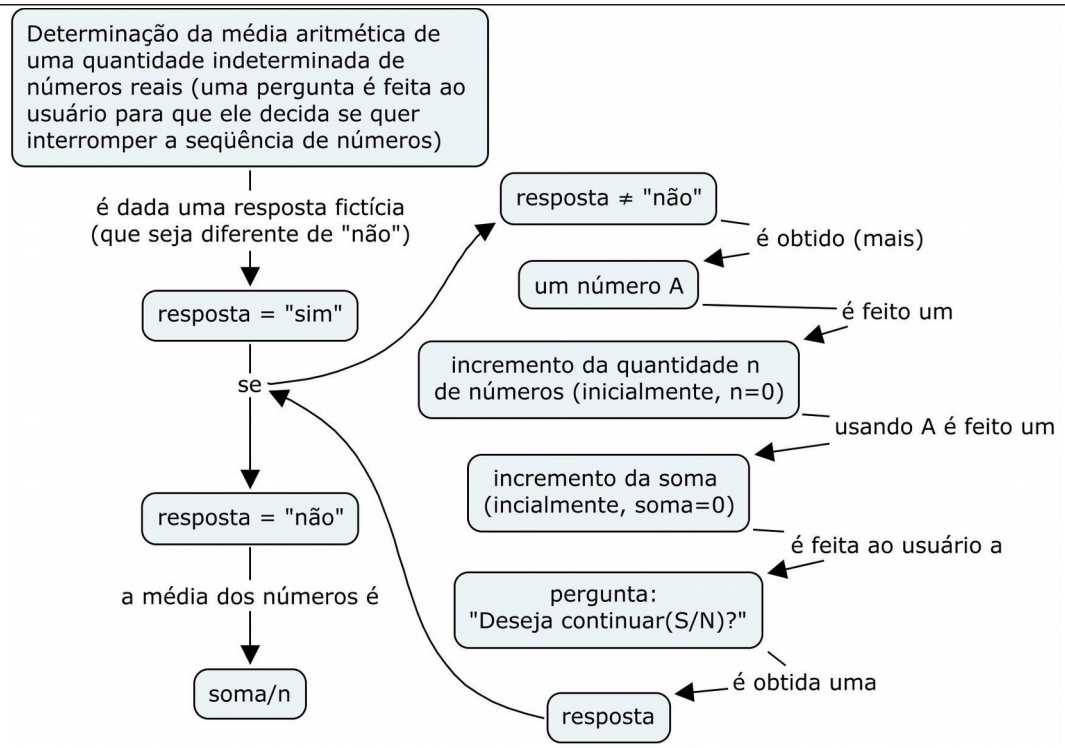
Interrupção externa aos dados. Existe uma variedade de maneiras de o usuário avisar o fim da leitura sem usar marcas nos dados. Para interromper a leitura, podemos utilizar, por exemplo, teclas especiais como a tecla *ESC* (não abordaremos este tema pois este nível de programação extrapola os objetivos dessa disciplina). Em programação de computadores, a interrupção externa se aplica a qualquer tipo de processo e não somente no caso de leitura de dados. No exemplo que estamos explorando (cálculo da média dos números), sem usar recursos avançados por enquanto, vamos ler os números um a um e verificando de há interesse em continuar. Embora não seja muito prático, podemos ler o primeiro número, o segundo, etc, e a cada leitura permitimos a opção de interromper o processo.

Vejamos então o exemplo abaixo.

Exemplo 3.13

Consideremos o seguinte enunciado: Calcular a média aritmética de uma quantidade indeterminada de números reais. Após cada leitura de número, uma pergunta é feita ao usuário para que ele decida se quer interromper. Após a pergunta "Deseja continuar (S/N)?", o usuário encerrará se responder com "N" ou "n".

Para o encaminhamento da solução, vemos que uma resposta do usuário tem que ser recepcionada e depois precisamos testá-la (se é um "sim" ou um "não"). Somente dessa maneira saberemos qual é o desejo do usuário. A solução está descrita no mapa abaixo:



No algoritmo, precisaremos de uma variável para representar a resposta do usuário. O espaço de apenas um caractere é suficiente para representar um "sim" (com um "`S`"), ou um "não" (com um "`N`"), em letras maiúsculas ou minúsculas. À resposta daremos o nome de `Resp`:

```
Algoritmo
1 n ← 0
2 soma ← 0.0
3 Resp ← "S"
4 enquanto (Resp ≠ "N") e (Resp ≠ "n")  faça:
4.1   escrever "Digite um número real:"
4.2   ler A
4.3   n ← n + 1
4.4   soma ← soma + A
4.5   escrever "Deseja continuar(S/N)?"
4.6   ler Resp
5 escrever "A média dos números é", soma/n
Fim-Algoritmo
```

Os passos 1, 2 e 3 do algoritmo acima são usados para inicializar respectivamente as variáveis `n`, `soma` e `Resp` . Sobre as variáveis `n` e `soma` já sabemos quais são os objetivos de suas existências e porque devem ser inicializadas. No caso da variável `Resp` a justificativa de sua atribuição inicial está no próximo passo que é a estrutura de repetição.

A estrutura do passo 4 está montada para repetir os comandos 4.1 a 4.6 enquanto o usuário **não** digitar "N" (ou "n") como resposta. Do modo como ela está posicionada no algoritmo, a decisão do usuário somente é conhecida no último comando do bloco (comando 4.6), mas o teste da condição acontece no início da estrutura. Como devemos estar percebendo, a atribuição do passo 3 é apenas um valor inicial (uma resposta fictícia do usuário) para permitir a realização do teste da condição da estrutura de repetição. Propositalmente colocamos um valor para `Resp` (e até poderia ser outro diferente de "`S`") que torne verdadeira a condição (`Resp` ≠ "`N`" e `Resp` ≠ "`n`") e haja sempre a primeira iteração.

O passo 5 corresponde à escrita do resultado e não há necessidade de verificar a divisão por zero. Isto acontece porque a primeira execução sempre ocorrerá fazendo `n` ser, no mínimo, igual a 1.

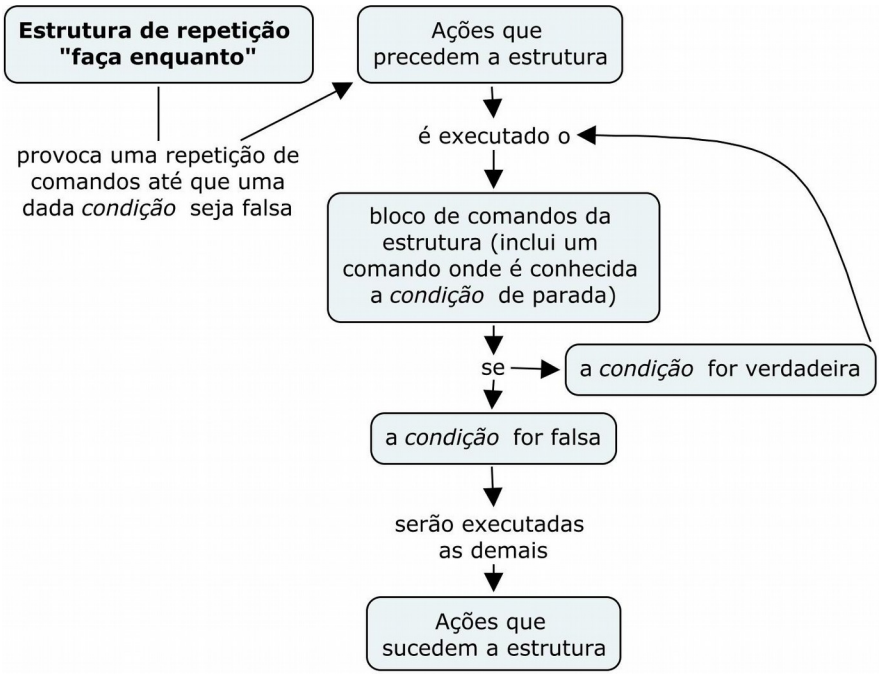
Vejamos mais uma vez um exemplo numérico calculando a média aritmética dos números `325.1`, `18.3` e `128.0` seguindo o quadro abaixo:

Passos do algoritmo:	É verdade que `Resp` ≠ "`N`" e `Resp` ≠ "`n`"?	Var. **A**	Var. **n**	Var. **soma**	Var. **Resp**	Observações:
1º. 2º. e 3º. passos	-	-	0	0.0	"S"	Inicializações de `n`, `soma` e `Resp`
4º. passo, 1ª. execução	"S" ≠ "N" e "S" ≠ "n" (é verdade). Ocorrerá uma repetição.	325.1	1	325.1	"S"	(4.1 e 4.2) leitura de A (4.3) n = 0 + 1 = 1 (4.4) soma=0.0 + 325.1=325.1 (4.5 e 4.6) leitura de Resp
4º. passo, 2ª. execução	"S" ≠ "N" e "S" ≠ "n" (é verdade). Ocorrerá mais uma repetição.	18.3	2	343.4	"S"	(4.1 e 4.2) leitura de A (4.3) n = 1 + 1 = 2 (4.4) soma=325.1+18.3=343.4 (4.5 e 4.6) leitura de Resp
4º. passo, 3ª. execução	"S" ≠ "N" e "S" ≠ "n" (é verdade). Ocorrerá mais uma repetição.	128.0	**3**	**471.4**	"**N**"	(4.1 e 4.2) leitura de A (4.3) n = 2 + 1 = 3 (4.4)soma=343.4+128.0=471.4 (4.5 e 4.6) leitura de Resp
-	"**N**" ≠ "N" e "**N**" ≠ "n" (é falso). Não mais correrá repetição.					-
Estado final da memória		128.0	**3**	**471.4**	"**N**"	

Para encerrar a entrada de dados, foi digitado o valor "**N**" (na terceira execução do passo 4) como resposta (valor de **Resp**) à pergunta: "`Deseja continuar(S/N)?`". O ciclo de repetições se encerrou porque a expressão "**N**" ≠ "N" e "**N**" ≠ "n" se tornou falsa. De posse dos valores finais de `n` (igual a 3) e `soma` (igual a 471.4) o valor da média aritmética é calculado e a saída é:

```
A média dos números é 157.13333
```

O ***Exemplo 3.13*** pode ser reescrito de maneira que não se precise da inicialização da variável `Resp`. Isto somente é possível se o teste da condição acontecer no fim da estrutura de repetição. Logo, precisaríamos de outro formato, denominado de estrutura *faça-enquanto*, cuja lógica pode ser descrita pelo mapa seguinte:



Nos algoritmos, essa estrutura pode ser escrita coma seguinte sintaxe:

```
faça:
    bloco de comandos
enquanto (condição)
```

Onde: A **condição** amarrada à estrutura é testada no fim da estrutura.

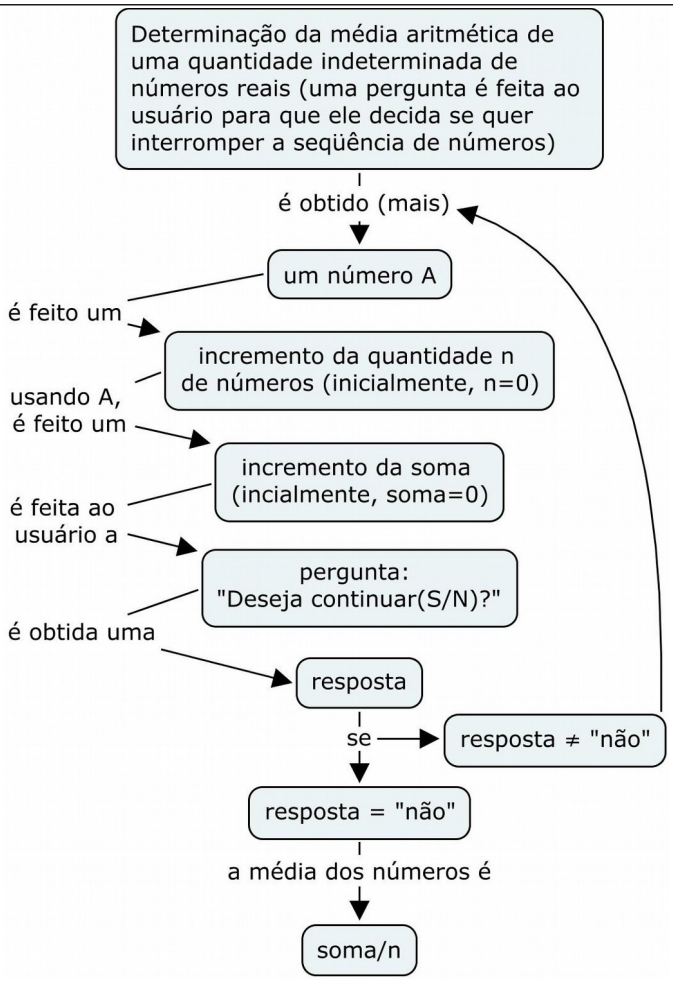
O **bloco de comandos** será executado repetidamente enquanto a condição estiver sendo atendida e a primeira iteração sempre ocorrerá.

Vejamos o exemplo seguinte.

O ***Exemplo 3.14*** seguinte é uma versão do ***Exemplo 3.13*** usando a estrutura *faça-enquanto*.

Exemplo 3.14

A solução abaixo usa a estrutura *faça-enquanto* (o teste no fim do bloco) para calcular a média de uma quantidade indeterminada de números reais. Desta vez, não há necessidade de inicialização da variável `Resp` porque a resposta do usuário é conhecida e testada somente no fim do bloco de comandos:




```
Algoritmo
1 n ← 0
2 soma ← 0.0
3 faça:
3.1   escrever "Digite um número real:"
3.2   ler A
3.3   n ← n + 1
3.4   soma ← soma + A
3.5   escrever "Deseja continuar(S/N)?"
3.6   ler Resp
  enquanto (Resp ≠ "N") e (Resp ≠ "n")
4 escrever "A média dos números é", soma/n
Fim-Algoritmo
```

Vejamos a seguir mais um problema em que teremos a oportunidade resolvê-lo usando a estrutura “enquanto” (com o teste no início do bloco de comandos) e a estrutura “faça-enquanto” (com o teste no fim).

Exemplo 3.15

Problema: Calcular e escrever o seno e o cosseno de um ângulo dado em graus, de acordo com a escolha do usuário. O algoritmo deve prever um menu com quatro opções: (1) Calcular o seno, (2) Calcular o cosseno, (3) Ler novo ângulo e (4) Encerrar.

A escolha feita pelo usuário (que será representada no algoritmo pela variável `opc`) somente é conhecida após o mesmo visualizar o menu. Isto significa, portanto, que esta escolha não será conhecida no início das iterações. Logo, para usar a estrutura "enquanto", é necessário que se faça uma inicialização da variável `opc` com um valor qualquer, desde que seja diferente da opção 4 (obviamente, porque este valor encerraria o processo). Analisemos então o algoritmo abaixo, onde o valor escolhido para inicializar `opc` é o zero e `ang` é a variável que conterá o valor do ângulo em graus:

```
Algoritmo
1 escrever "Forneça a medida de um ângulo em graus: "
2 ler ang
3 opc ← 0
4 enquanto(opc ≠ 4):
4.1   escrever "  (1)Calcular o seno"
4.2   escrever "  (2)Calcular o cosseno"
4.3   escrever "  (3)Novo ângulo"
4.4   escrever "  (4)Encerrar"
4.5   escrever "Digite sua opção > "
4.6   ler opc
4.7   se (opc = 1) então:
            escrever "O seno de",ang,"graus é", seno(ang*pi/180.0)
      senão se (opc = 2) então:
            escrever "O cosseno de",ang,"graus é",cos(ang*pi/180.0)
      senão se (opc = 3) então:
            escrever "Forneça novo ângulo: "
            ler ang
      senão se (opc = 4) então:
            escrever "Programa encerrado! "
      senão:
            escrever "Opção inválida! "
Fim-Algoritmo
```

Neste algoritmo, nos passos 1, 2 e 3, nessa sequência, acontecem a leitura do ângulo `ang` em graus e a inicialização da variável `opc` que armazenará a opção do usuário. O passo 4 é a estrutura de repetição que, enquanto o usuário não desejar encerrar (digitando opc = 4), executará os comandos 4.1 a 4.7. Os comandos 4.1 a 4.4 escrevem o menu de opções, o passos 4.5 e 4.6 servem para ler a opção do usuário e o passo 4.7 é o encadeamento de

estruturas de seleção com o objetivo executar a ação escolhida pelo usuário. Apenas uma das ações é executada.

Nas duas primeiras opções disponíveis ocorrem os cálculos do seno e do cosseno. Considerando-se que as linguagens de programação normalmente adotam medidas de ângulos em radianos, o algoritmo já prevê esta conversão (multiplicando `ang` pelo número π (PI) e dividindo por 180). Na opção 3, há uma nova leitura do ângulo, substituindo o valor antigo de `ang`. Se o usuário escolher a opção 4, nenhum comando relevante será executado, apenas se limitando a emitir uma mensagem (que seria até dispensável!). Mas, esta opção está presente no encadeamento por ser necessária no ponto de vista da comunicação com o usuário. Explicando, se ela não for incluída e o usuário desejar encerrar o programa, a mensagem "Opção inválida!" seria escrita na tela. A mensagem não refletiria a verdade porque o 4 tem uma utilidade que é, justamente, a de indicar o encerramento do processo.

Teremos a seguir a solução do mesmo problema do exemplo acima, desta vez usando a estrutura "faça-enquanto".

Exemplo 3.16

O problema a ser resolvido aqui é o mesmo do ***Exemplo 3.15***: Calcular e escrever o seno e o cosseno de um ângulo dado em graus, de acordo com a escolha do usuário e o algoritmo deve prever um menu com as quatro opções já citadas.

A escolha do usuário dentre as opções disponíveis não é conhecida no início das iterações. Logo, para usar a estrutura "faça-enquanto", não é necessária a inicialização da variável `opc`. Porque o teste da condição (que envolve a variável `opc`) somente acontece no fim do bloco de comandos. Analisemos então o algoritmo a seguir:

```
Algoritmo
1 escrever "Forneça a medida de um ângulo em graus: "
2 ler ang
3 faça:
3.1   escrever "  (1)Calcular o seno"
3.2   escrever "  (2)Calcular o cosseno"
3.3   escrever "  (3)Novo ângulo"
3.4   escrever "  (4)Encerrar"
3.5   escrever "Digite sua opção > "
3.6   ler opc
3.7   se (opc = 1) então:
            escrever "O seno de",ang,"graus é", seno(ang*pi/180.0)
      senão se (opc = 2) então:
            escrever "O cosseno de",ang,"graus é",cos(ang*pi/180.0)
      senão se (opc = 3) então:
            escrever "Forneça novo ângulo: "
```

```
            ler ang
      senão se (opc = 4) então:
            escrever "Programa encerrado! "
      senão:
            escrever "Opção inválida! "
  enquanto(opc ≠ 4)
Fim-Algoritmo
```

Podemos observar comparando o algoritmo logo acima com o algoritmo do ***Exemplo 3.15***, que a modificação foi mínima. A decisão por adotar uma das duas versões praticamente reside no fato de se querer inicializar a variável `opc` ou não. Geralmente é esta a decisão a ser tomada quando o assunto é a escolha entre as duas formas: usar a estrutura que precisa de inicialização da variável envolvida na condição ou usar a estrutura que não precisa desta inicialização. A rigor, não há uma exigência técnica a respeito, que obrigue a escolha de uma ou outra. É costume se escolher estas formas de acordo com a melhor facilidade no momento da implementação e de acordo com as características da linguagem adotada.

Laboratório - Estrutura de repetição "enquanto"

Objetivos

Conferir o uso da estrutura de repetição “enquanto”, dado que a condição de interrupção das iterações independe do conhecimento sobre a quantidade destas.

Usar os recursos disponíveis na linguagem Python para aplicar os conceitos e testar os algoritmos desta subunidade.

Recursos e experimentação

Podemos usar uma estrutura de repetição “enquanto” nas situações em que se precisa repetir a execução de um bloco de comandos, uma quantidade indeterminada de vezes.

Sintaxe. A linguagem Python implementa esta estrutura com a seguinte sintaxe:

```
while (condição):
    bloco de comandos
```

onde a **`condição`** é dada por uma expressão lógica, uma constante ou uma variável do tipo lógico. O **`bloco de comandos`** deve respeitar a endentação já discutida anteriormente, inclusive, se este se reduzir a apenas um comando, pode ser escrito adiante do símbolo “**:**”.

As repetições acontecerão enquanto a condição for verdadeira. A condição é testada primeiramente e somente depois desse teste é que a estrutura fará mais uma iteração ou não.

Experimento 01

O programa a seguir é uma implementação do ***Exemplo 3.12***. Calcula a média aritmética de uma quantidade indeterminada de números reais, onde o último número, que não entrará nos cálculos, é o 999 (o flag). O primeiro número é lido e, se não for o flag, o programa prosseguirá com a leitura repetindo esse teste sempre **antes** de contar e somar cada número lido.

Criar o arquivo `L322_01.py` e digitar as seguintes linhas de código:

```
n = 0
soma = 0.0
print ('Digite os números (999 - encerra)')
A = float(input())
while A != 999:
    n+=1
    soma += A
    A = float(input())
if n!=0: print ('A média dos números é', soma/n)
else: print ('Nenhum número foi digitado')
```

Execução de `L322_01.py`. Calcularemos a média aritmética dos números: `325.1`, `18.3` e `128.0` (como já demonstramos anteriormente). Para tanto, executamos o programa, digitamos os números desejados, e, para informar ao programa que são apenas esses três valores, digitamos o *flag* 999. Então, obtemos o seguinte:

```
>>>
Digite os números (999 - encerra)
325.1
18.3
128.0
999
A média dos números é 157.133333333
>>>
```

Se digitarmos o *flag* em primeiro lugar, este dado é “entendido” pelo programa como a informação de que não há números para se calcular a média aritmética. Executando o programa e digitando o *flag* visualizamos:

```
>>>
Digite os números (999 - encerra)
999
Nenhum número foi digitado
>>>
```

Experimento 02

O ***Exemplo 3.13*** demonstra o uso da estrutura “enquanto” quando o objeto de teste tem seu conteúdo conhecido somente no final do bloco de comandos. O programa a seguir implementa aquele exemplo, calculando a média de uma quantidade indeterminada de números reais, permitindo, a cada leitura de dado, que o usuário interrompa o processo. O instrumento que avisa ao programa para fazer que a interrupção não faz parte dos dados (não é um número). O usuário deve responder com “N” ou “n” à pergunta "Deseja continuar (S/N)?", se pretender parar a repetição, e sua resposta somente é conhecida depois que o bloco de comandos tem sua execução iniciada.

Criar o arquivo `L322_02.py` e digitar as seguintes linhas de código:

```
n = 0
soma = 0.0
Resp = 'S'
while (Resp != 'N') and (Resp != 'n'):
    A = float(input('Digite um número real: '))
    n+=1
    soma += A
    Resp = input('Deseja continuar(S/N)? ')
print ('A média dos números é', soma/n)
```

Execução de `L322_02.py`. Iremos experimentar mais uma vez com os mesmos números: `325.1`, `18.3` e `128.0`. Executamos o programa e digitamos os números seguidamente, mas temos que responder à pergunta do programa a cada número digitado, como segue:

```
>>>
Digite um número real: 325.1
Deseja continuar(S/N)? s
Digite um número real: 18.3
Deseja continuar(S/N)? s
Digite um número real: 128.0
Deseja continuar(S/N)? n
A média dos números é 157.133333333
>>>
```

Observação: Usando as propriedades da linguagem Python sobre listas, podemos reescrever a expressão `(Resp != 'N') and (Resp != 'n')`. Esta condição significa uma verificação se `Resp` é um caractere diferente de `'N'`, maiúsculo ou minúsculo (se for igual, o processo para). Para dizer em Python que `Resp` é um dos valores “N” ou “n” é bastante escrever a expressão: `Resp` **`in`** `['N', 'n']`, ou seja, o conteúdo de `Resp` pertence à lista `['N', 'n']`. Para dizer que não pertence escrevemos a negativa da expressão anterior, `not (Resp in ['N', 'n'])`, que é equivalente a: `Resp` **`not in`** `['N', 'n']`.

Experimento 03

Este experimento tem o objetivo apenas de mostrar a equivalência entre as expressões lógicas: `(Resp != 'N') and (Resp != 'n')` e `(Resp not in ['N', 'n'])`.

Criar o arquivo `L322_03.py` e digitar as seguintes linhas de código, a partir da modificação do programa `L322_02.py`:

```
n = 0
soma = 0.0
Resp = 'S'
while Resp not in ['N', 'n']:
    A = float(input('Digite um número real: '))
    n+=1
    soma += A
    Resp = input('Deseja continuar(S/N)? ')
print ('A média dos números é', soma/n)
```

Execução de `L322_03.py`. Experimentando com os mesmos números do experimento anterior temos (um resultado idêntico):

```
>>>
Digite um número real: 325.1
Deseja continuar(S/N)? s
Digite um número real: 18.3
Deseja continuar(S/N)? s
Digite um número real: 128.0
Deseja continuar(S/N)? n
A média dos números é 157.133333333
>>>
```

Experimento 04

Nesse experimento faremos da implementação do algoritmo do ***Exemplo 3.14***. Este algoritmo resolve o mesmo prob***lema do*** Exemplo 3.13, mas usa a estrutura "faça-enquanto", ou "enquanto" com o teste da condição no fim do bloco de comandos.

A linguagem Phyton não possui estrutura especial para implementar o "faça-enquanto". Quando esta estrutura for recomendável para a solução de um determinado problema, podemos fazer uma adaptação da estrutura *while* ("enquanto") da seguinte maneira. Como esta última exige um teste no início dos comandos, usamos uma repetição de condição constante igual a `True` (e, em princípio, uma repetição infinita – comumente chamada de "loop infinito") e preparamos um critério para interrupção no fim, usando a palavra reservada **`break`**.

Precisamos apenas substituir a lógica condição da estrutura, trocando:

"Fazer enquanto" **`Resp not in ['N', 'n']`**

pela equivalente,

"Fazer até que" **`Resp in ['N', 'n']`**

Criar o arquivo `L322_04.py` e digitar as seguintes linhas de código:

```
n = 0
soma = 0.0
while True:
    A = float(input('Digite um número real: '))
    n+=1
    soma += A
    Resp = input('Deseja continuar(S/N)? ')
    if Resp in ['N', 'n']: break
print ('A média dos números é', soma/n)
```

Observamos no programa acima que a linha de código "`if Resp in ['N', 'n']: break`", posicionada no fim do bloco, garante que as repetições irão acontecer até que o conteúdo de `Resp` seja um dos elementos da lista `['N', 'n']`. Se isto for verdade, o `break` irá interromper as iterações, pois, em princípio, a linha de código "`while True:`" faria com que o número de repetições fosse infinito.

Execução de `L322_04.py`. A execução desse programa produz uma saída idêntica às saídas produzidas pelos programas dos experimentos anteriores, ***Experimento 02*** e ***Experimento 03***, e, portanto, dispensa sua transcrição aqui.

Os dois experimentos seguintes implementam soluções de um mesmo problema. Teremos a oportunidade de comparar as implementações dos algoritmos dados no ***Exemplo 3.15*** e no ***Exemplo 3.16***, observando qual das versões pode ser mais prática para ser implementada na linguagem Python.

Experimento 05

O programa seguinte implementa o algoritmo do ***Exemplo 3.15***.

Criar o arquivo `L322_05.py` e digitar as seguintes linhas de código:

```
from math import*
ang = float(input('Forneça a medida de um ângulo em graus: '))
opc = 0 #um valor qualquer diferente de 4 (pois 4 encerraria o
programa)
while(opc!=4):
    print ('  (1)Calcular o seno')
    print ('  (2)Calcular o cosseno')
    print ('  (3)Novo ângulo')
```

```
    print ('  (4)Encerrar')
    opc = int(input('Digite sua opção > '))
    if opc==1:print('O seno de',ang,'graus é',sin(radians(ang)))
    elif opc==2:print('O cosseno de',ang,'graus é',cos(radians(ang)))
    elif opc==3: ang = float(input('Forneça novo ângulo: '))
    elif opc==4: print ('Programa encerrado!')
    else: print ('Opção inválida!')
```

Observação: Utilizamos nesse programa a função `radians(ang)` do módulo `math` que efetua prontamente o cálculo: `ang*pi/180.0`.

Execução de `L322_05.py`. Executando o programa, digitando o valor 60 para o ângulo e a opção 2 do menu, obtemos:

```
>>>
Forneça a medida de um ângulo em graus: 60
  (1)Calcular o seno
  (2)Calcular o cosseno
  (3)Novo ângulo
  (4)Encerrar
Digite sua opção > 2
O cosseno de 60 graus é 0.5
  (1)Calcular o seno
  (2)Calcular o cosseno
  (3)Novo ângulo
  (4)Encerrar
Digite sua opção >
```

Vemos, como tínhamos previsto, que o menu se repete a cada iteração da estrutura “while”. Após o usuário digitar sua a opção, o programa exibe o resultado e mostra novamente o menu. Com o programa ainda em execução, se quisermos agora mudar para o ângulo de 45 graus, é só digitar a opção 3. Fazendo isto, obtemos:

```
...
Digite sua opção > 3
Forneça novo ângulo: 45
  (1)Calcular o seno
  (2)Calcular o cosseno
  (3)Novo ângulo
  (4)Encerrar
Digite sua opção >
```

A partir de então o ângulo corrente é 45 graus, que ficará à disposição para o cálculo do seno ou do cosseno. Usando o menu, podemos também trocar mais uma vez de ângulo ou encerrar o programa. Considerando esta última opção, digitando o 4 teremos:

```
...
Digite sua opção > 4
Programa encerrado!
>>>
```

Dessa maneira, a sessão do programa se encerra emitindo uma mensagem (o menu não mais é exibido).

Experimento 06

O problema tratado neste experimento é exatamente o mesmo do experimento anterior, mas aqui é implementada a estrutura "faça-enquanto". O algoritmo é o do ***Exemplo 3.16***. Da mesma maneira como fizemos ***no*** Experimento 04, para implementar esta estrutura em Python, faremos um "loop infinito" com o "while" e substituímos a lógica da condição por uma lógica equivalente. Ou seja:

"Fazer enquanto" **`(opc != 4)`**

é equivalente a:

"Fazer até que" **`(opc == 4)`**

Daí, esta nova lógica, que é aplicada no final do bloco de comandos, toma a forma de um teste que, se for verdadeiro, interrompe as repetições. Sabendo-se que esse teste é

`if (opc == 4): break`

e que ele já existe no encadeamento, é bastante aproveitá-lo. Este aproveitamento se dá somente acrescentando mais um comando, o `break`, ao bloco de comandos que testa a opção 4 (onde já existe o comando "`print 'Programa encerrado!'`").

Criar o arquivo `L322_06.py` e digitar as seguintes linhas de código:

```
from math import*
ang = float(input('Forneça a medida de um ângulo em graus: '))
while(True):
    print ('  (1)Calcular o seno')
    print ('  (2)Calcular o cosseno')
    print ('  (3)Novo ângulo')
    print ('  (4)Encerrar')
    opc = int(input('Digite sua opção > '))
    if opc==1:print ('O seno de',ang,'graus é', sin(radians(ang)))
    elif opc==2:print('O cosseno de',ang,'graus é',cos(radians(ang)))
    elif opc==3: ang = float(input('Forneça novo ângulo: '))
```

```
    elif opc==4:
        print ('Programa encerrado!')
        break
    else: print ('Opção inválida!')
```

Execução de `L322_06.py`. Pode-se mostrar que a saída produzida e a interação deste programa com o usuário são idênticas a aquelas do programa `L322_05.py`. O leitor deve usar os mesmos valores testados no ***Experimento 05*** e conferir esta afirmativa.

Exercício de autoavaliação

Elabore os programas abaixo com base nos conhecimentos construídos nesta subunidade.

1 - Faça um programa em Python para implementar o algoritmo elaborado no quinto item do exercício de autoavaliação da **Subunidade 1.2.2** (que trata do cálculo da soma de uma quantidade indeterminada de números diferentes de 0-zero) ;

2 - Elabore uma nova versão do programa da questão anterior de modo que também determine e escreva o maior e o menor valor lido;

3 - Elabore uma nova versão do programa do item 3 do exercício de autoavaliação da **Subunidade 3.1.3** (ler a nota de um aluno e escrever o conceito correspondente) de modo que o processo seja repetido até que o usuário digite 'N' como resposta à pergunta: “Continuar? (S/N)”.

MÓDULO IV

Estruturas de Dados

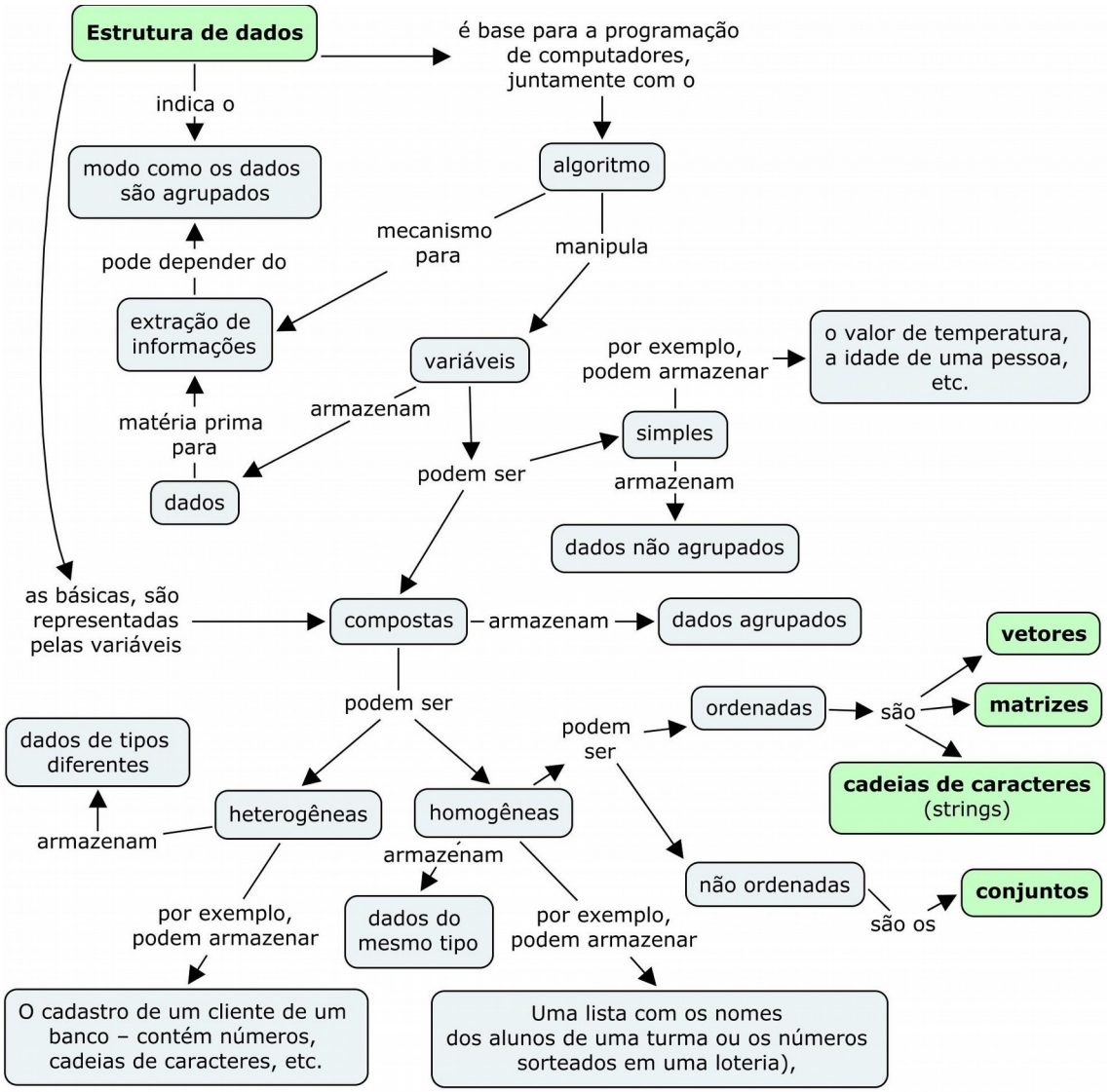
Na **Unidade II.1**, tivemos a oportunidade de tratar dos tipos básicos de dados manipuláveis pela máquina. Ocorre que a extração de informações pode depender fortemente de certa "organização" que os dados devam assumir. Explicitamente, determinadas informações somente poderão ser extraídas se os dados fizerem parte de uma estrutura apropriada.

Em uma primeira análise, afirmamos que uma *estrutura de dados* é uma maneira de tratar os dados de forma agrupada. Tivemos uma antecipada experiência com um tipo de agrupamento de dados ao abordarmos as cadeias de caracteres na **Subunidade 2.1.1**. Contudo, o que fizemos fundamentalmente até este momento foi tratar o dado básico isoladamente, cada um com seu tipo e fazendo ocupar posições independentes de memória. Ou seja, trabalhamos com dados *e* variáveis simples. Neste módulo, pretendemos avançar sobre novos tipos de informações aplicando a técnica de referenciar blocos de dados na memória do computador usando um único nome. São as chamadas *variáveis compostas*.

Se, por exemplo, nos referirmos a um dado sobre temperatura ou a idade de uma pessoa de maneira distinta, precisaremos apenas de variáveis simples para tanto. Todavia, se tivermos um cadastro de um cliente de um banco (com nome, endereço, idade, etc.), uma lista com os nomes dos alunos de uma turma ou um conjunto de números sorteados em uma loteria, por exemplo, não podemos deixar de tratá-los em blocos e, portanto, precisaremos de variáveis compostas.

As variáveis compostas, por sua vez, podem reunir dados do mesmo tipo ou não. Quando uma variável composta agrupa dados somente do mesmo tipo (como, por exemplo, uma lista com os nomes dos alunos de uma turma ou os números sorteados em uma loteria), dizemos que é uma *variável composta homogênea*. Se assim não o for (como, por exemplo, o cadastro de um cliente de um banco – contém números, cadeias de caracteres, etc.), será uma *variável composta heterogênea*. Estudaremos apenas as variáveis compostas homogêneas, que é o assunto deste módulo.

Os conceitos acima descritos podem ser representados no mapa abaixo:



Objetivos

- Conceituar estrutura de dados
- Identificar as estruturas de dados homogêneas e os casos de aplicação das mesmas

Unidades

- Unidade IV.1 - Estruturas homogêneas básicas - Vetores e Matrizes
- Unidade IV.2 - Estruturas homogêneas especiais - Cadeias de Caracteres e Conjuntos

Unidade IV.1

Estruturas homogêneas básicas – Vetores e Matrizes

As variáveis compostas homogêneas podem ser vistas como lugares reservados na memória do computador para conter "pacotes" de valores, todos do mesmo tipo. Estes pacotes podem estar em uma dimensão (ou seja, enfileirados) e marcados com índices. Sob esta forma, cada pacote será chamado de *vetor*. Localizar um elemento num vetor significa apenas informar o índice (um valor numérico) de sua posição no referido vetor. Quando os dados estiverem organizados em mais de uma dimensão, teremos uma estrutura de dados chamada *matriz*. Por exemplo, dispondo de uma matriz bidimensional, localizamos um elemento desta determinando os índices, linha e coluna, de onde o mesmo se encontra.

Na unidade em curso, trataremos destas estruturas: vetores e matrizes. As cadeias de caracteres possuem características de vetores, porém, em particular, encerram um significado contextual e assim, serão estudadas como estruturas especiais na próxima unidade.

4.1.1 Vetores

Define-se um vetor através do seu nome e do seu tamanho em termos da quantidade de elementos. Muitas vezes, o tamanho do vetor é atribuído antes de se conhecer a real quantidade de dados a serem armazenados e esse tamanho não muda até o término da execução do programa (alocação *estática*). A modificação do tamanho do vetor durante a execução do programa é permitida quando a alocação é *dinâmica*. Podemos fazer isso com facilidade na linguagem Python, por exemplo.

Nos algoritmos, genericamente, nos referiremos a um vetor com a seguinte notação:

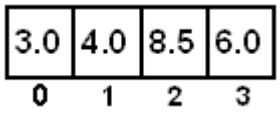
tipo_básico **nome_do_vetor [***tamanho***]**.

Os elementos do vetor são de um determinado *tipo_básico* e guardam suas posições rigorosamente. A localização de cada um na estrutura é feita através de índices. Estes índices podem variar de representação, desde que sejam valores discretos. Aqui, adotaremos a sequência dos números inteiros não negativos (a partir de zero). Isto é, segundo esse critério, para um vetor de um dado *tamanho* (isto é, com *tamanho* significando a quantidade de componentes), os índices irão variar de *0* até *tamanho-1*. Por exemplo, num vetor de tamanho igual a 10 (dez), seus índices variam de acordo com a sequência {0, 1, 2,..., 9}.

Exemplo 4.1

Vamos supor que as notas possíveis de um aluno durante um semestre letivo são: nota do primeiro bimestre, nota do segundo bimestre, nota de reavaliação e nota da prova final. Podemos criar o vetor `nota` para armazenar as notas nessa ordem, usando os índices 0, 1, 2 e

3 para localizá-las na estrutura (nos algoritmos, este seria o vetor `numérico nota[4]` , ou seja, a variável é `nota` e tem quatro lugares para conter números). Consideremos um aluno que tirou as notas 3.0, 4.0, 8.5 e 6.0, respectivamente. Graficamente, podemos indicar o vetor `nota` a partir de um modelo de disposição na memória como o seguinte:



O acesso a cada nota é feito através do índice respectivo. Uma referência à nota do primeiro bimestre é feita por `nota[0]`, a nota do segundo bimestre é `nota[1]`, a reavaliação é `nota[2]` e a final é `nota[3]`.

Atribuição de valores

Após sua criação, da mesma maneira que as variáveis simples, um vetor pode ter seus valores preenchidos tanto por atribuição quanto por leitura da unidade de entrada. A atenção deve estar na referência individual aos elementos.

Exemplo 4.2

Tomemos a variável `nota` do ***Exemplo 4.1***. O citado vetor pode ser preenchido atribuindo-se valores cada um dos seus componentes, da seguinte maneira:

```
nota[0] ← 3.0
nota[1] ← 4.0
nota[2] ← 8.5
nota[3] ← 6.0
```

Um vetor pode ser inicializado. Esta inicialização consiste na atribuição de valores desde seu "nascimento" na memória do computador. Numa linguagem em que não se precisa de declaração, a inicialização é obrigatória, mas o tamanho do vetor continua sendo aquele determinado na inicialização. A variável `nota` dos exemplos acima pode ser inicializada como:

```
nota ← [3.0, 4.0, 8.5, 6.0]
```

(o valor de cada componente será atribuído obedecendo esta sequência).

Leitura e escrita

O preenchimento de um vetor por leitura da unidade de entrada também não difere muito da leitura de uma variável simples (pois cada componente do vetor pode ser vista como uma variável simples). Por exemplo, se quiséssemos trocar a atual nota de reavaliação (que é a nota de índice 2 do vetor) do aluno do ***Exemplo 4.1*** por outra digitada pelo usuário, seria bastante fazer:

```
ler nota[2]
```

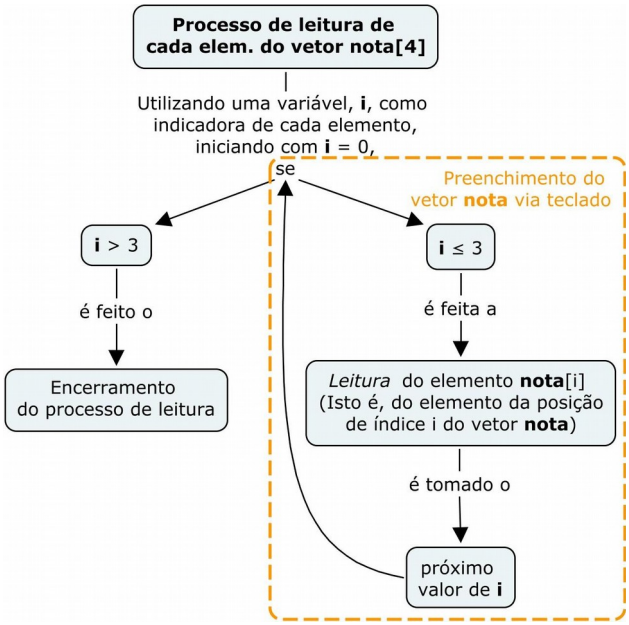
Se tivéssemos que ler todas as quatro notas, este comando teria que ser repetido para cada componente. No citado exemplo não é tão problemático repetir comandos, afinal ocuparia apenas quatro linhas:

```
ler nota[0]
ler nota[1]
ler nota[2]
ler nota[3]
```

Percebemos que não é sensato usar tal sequência de comandos, pois não usamos os recursos do computador ao nosso favor. Além disso, se tivéssemos bem mais que quatro componentes, não seria muito prático (e, talvez, inviável) repetir uma grande quantidade de linhas. Logo, a solução é usar uma estrutura de repetição. Podemos representar este processo no mapa abaixo.

Exemplo 4.3

Continuando com a exploração do ***Exemplo 4.1***, uma maneira de ler todas as quatro notas está representada no mapa abaixo:



Observamos no trecho grifado "Preenchimento do vetor `nota` via teclado" que a leitura de cada elemento do vetor `nota` ocorre seguindo a sequência dos índices: 0, 1, 2, 3. Ao mapa então se associa a seguinte estrutura:

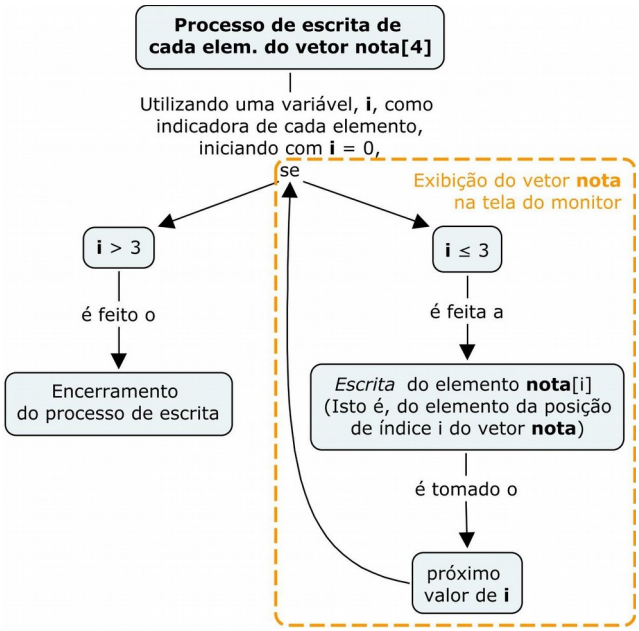
```
para i ← 0...3, faça:
    ler nota[i]
```

Isto é, o comando `ler nota[i]` será repetido para todos os valores da variável `i`, de 0 até 3 (ou seja, na primeira iteração, o comando é `ler nota[0]`, na segunda, é `ler nota[1]`, e assim por diante).

O raciocínio para leitura de cada elemento do vetor é também válido para a exibição individual dos elementos na tela do monitor. Ou seja, a maneira geral de escrever os componentes de um vetor na saída é usando uma estrutura de repetição.

Exemplo 4.4

Para escrever todas as notas do aluno do ***Exemplo 4.1***, usamos procedimento análogo ao da leitura conforme mapa abaixo:



A estrutura abaixo representa o processo descrito acima (observar "Exibição do vetor `nota` na tela do monitor"):

```
para i ← 0...3, faça:
    escrever nota[i]
```

(ou seja, essa estrutura substitui os comandos `escrever nota[0]`, `escrever nota[1]`, `escrever nota[2]`, `escrever nota[3]`).

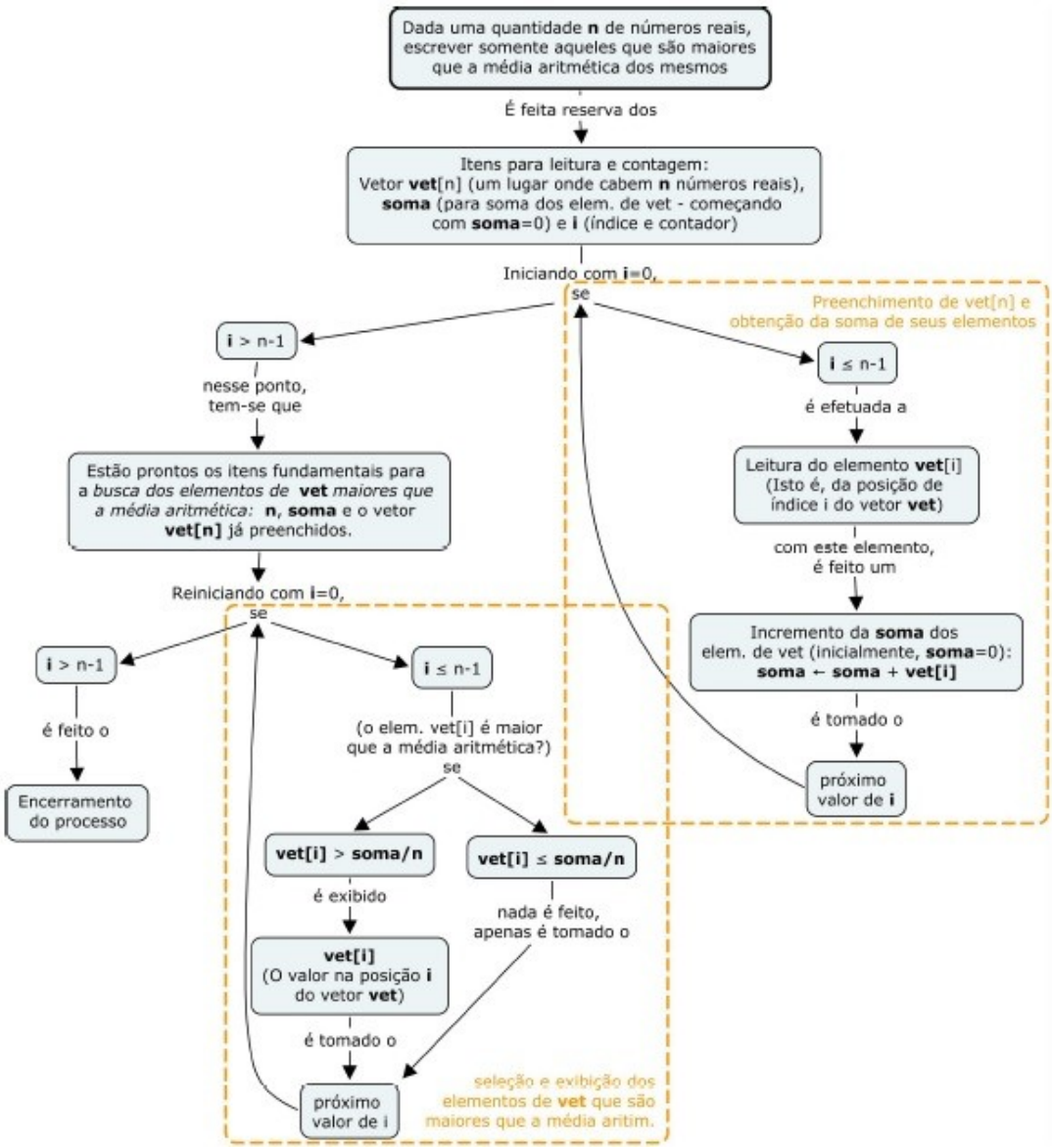
Veremos em seguida exemplos de uso de vetores para solução de problemas. Inicialmente, retornemos ao ***Exemplo 3.7*** (da **Unidade III.2**) em que mostramos um algoritmo para calcular a média aritmética de uma dada quantidade de números reais. O cálculo da média exigiu antes o cálculo da soma dos números, mas não foi necessário armazenar todos os números lidos. Então, cada número lido, depois de somado, era descartado na próxima leitura (todos os números foram lidos usando a mesma variável).

Se, estendendo o caso citado acima, solicitarmos que sejam listados todos os números maiores que a média aritmética deles mesmos, o problema passa a precisar de nova estratégia para a solução. A razão é simples: ora, se no algoritmo do ***Exemplo 3.7*** os números lidos eram descartados a cada nova leitura, como saber agora quais eram os números maiores que a média? Agora, sim, precisaremos guardar todos os números.

Exemplo 4.5

Problema: Ler uma dada quantidade de números reais e escrever somente aqueles que são maiores que a média aritmética dos mesmos.

Para calcular a média, precisamos da quantidade dos mesmos e da soma destes. Porém, para voltar e determinar os maiores que a média, precisamos armazenar todos os números. É aqui que precisamos de uma estrutura de dados, no caso, de um vetor. A estratégia é a seguinte. À medida que forem lidos, os números serão armazenados no vetor e acumulados em uma soma. De posse desta soma e da quantidade `n`, calculamos a média aritmética. Em seguida, percorremos todos os valores armazenados no vetor comparando cada um com a média e aquele que for maior que a mesma é escrito na tela. Chamaremos o vetor com `n` números reais de `vet[n]`. Vejamos esse processo representado no mapa conceitual abaixo:



Ao mapa acima se pode associar o algoritmo seguinte:

```
Algoritmo
1 escrever "Digite a quantidade de números"
2 ler n
3 criar vet[n]
4 soma ← 0.0
5 escrever "Digite os números"
6 para i ← 0...n-1, faça:
      ler vet[i]
      soma ← soma + vet[i]
7 escrever "Média aritmética:", soma/n
8 escrever "Valor(es) acima da média:"
9 para i ← 0...n-1, faça:
      se (vet[i] > soma/n) então:
            escrever vet[i]
Fim-Algoritmo
```

Os passos 1 e 2 são destinados a leitura da quantidade de números a serem lidos (o primeiro é apenas uma solicitação ao usuário). Deve ser lembrado que, a partir desse ponto do algoritmo, tudo fará sentido somente se o usuário digitar `n` diferente de zero. Vale o mesmo cuidado que tivemos em outros casos, fazendo-se um teste e deixando o algoritmo prosseguir somente se `n` for válido. Não o faremos aqui apenas para colocar a atenção em outros pontos ora discutidos.

O passo 3 corresponde à criação de `vet[n]`. Dependendo da linguagem em que for implementado o algoritmo, a criação pode ser através de uma declaração ou através de uma inicialização (normalmente, preenchendo todo vetor com zeros). O passo 4 é a inicialização da variável `soma` e o passo 5 é a exibição de uma mensagem solicitando do usuário que digite os números. No passo 6, a estrutura de repetição faz a leitura de cada elemento do vetor e acumula este valor na variável `soma`.

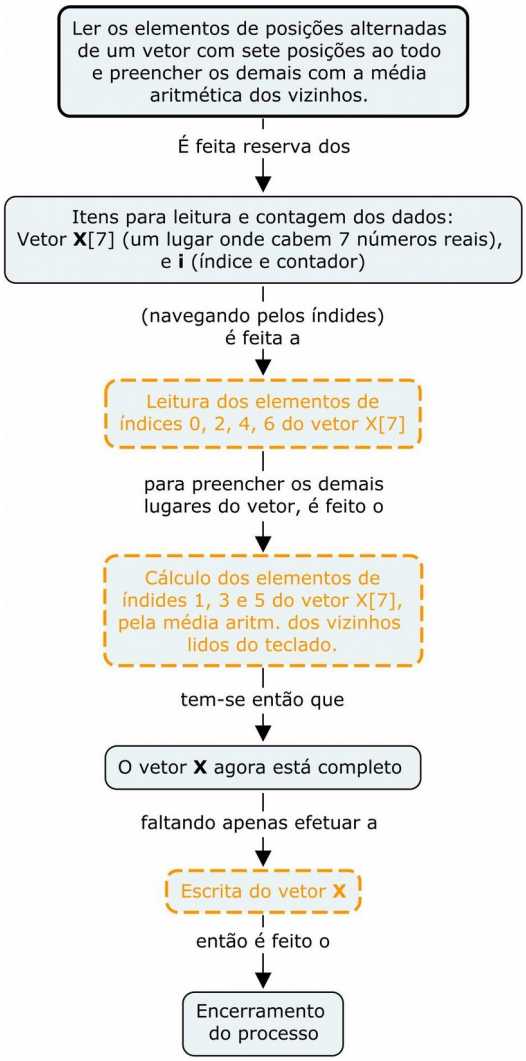
Continuando, com o valor de `soma` e de `n` é calculada e escrita a média aritmética no passo 7. Então, o passo 8 anuncia e o passo 9 escolhe os elementos e escreve o resultado. A estrutura de repetição (passo 9) percorre cada elemento do vetor comparando com a média aritmética e, toda vez que for maior que a média, o elemento será escrito na tela.

O uso de vetores pode ser mais direto e podemos até combinar algo que está em mais de um vetor, fazer operações com índices para selecionar elementos, etc. Vejamos o exemplo seguinte.

Exemplo 4.6

Problema: Ler alternadamente os elementos de um vetor de 7 (sete) elementos, começando a leitura a partir do primeiro. Cada componente que não foi lido deverá ser preenchido com a média aritmética de seus vizinhos. Ao final, escrever o vetor completo.

Chamemos de `X` o vetor a ser construído. O mapa abaixo mostra como o processo pode ser encaminhado:



Ou seja, serão lidos os componentes `X[0]`, `X[2]`, `X[4]` e `X[6]` (tal como os elementos hachurados na figura abaixo) e os elementos intermediários (`X[1]`, `X[3]` e `X[5]`) devem ser preenchidos a partir de seus vizinhos segundo a regra do problema. Ou seja, `X[1]=(X[0]+X[2])/2`, `X[3]=(X[2]+X[4])/2` e `X[5]=(X[4]+X[6])/2`.

No mapa acima, os conceitos de leitura, escrita e cálculo de elementos do vetor envolvem o acesso a cada elemento individualmente navegando-se pelos índices: Os índices dos elementos lidos são os números pares de 0 a 6 e os índices dos calculados são os ímpares 1, 3 e 5. Esse detalhamento pode ser visto no mapa e no correspondente algoritmo mostrados a seguir.

```
Algoritmo
1 criar X[7]
2 escrever "Digite os elementos do vetor alternadamente:"
```

```
3 i ← 0
4 enquanto (i < 7) faça:
      ler X[i]
      i ← i + 2
5 i ← 1
6 enquanto (i < 6) faça:
      X[i] ← (X[i-1] + X[i+1])/2
      i ← i + 2
7 escrever "Vetor completo:"
8 para i ← 0...6, faça:
      escrever X[i]
Fim-Algoritmo
```

Após a criação do vetor `X` na memória do computador (no passo 1) e solicitação ao usuário para digitação dos valores (no passo 2) o algoritmo continua com mais três tarefas: a leitura dos valores alternados, o cálculo dos valores intermediários e a escrita do vetor completo.

A leitura dos valores acontece nos passos 3 e 4. O passo 3 é a atribuição do valor inicial zero para `i` com o objetivo de construir os números pares de 0 até 6. O passo 4 é uma estrutura de repetição que, a cada iteração, lê o elemento `X[i]` (na posição `i` do vetor) e, em seguida, faz um incremento de 2 na variável `i`. Logo, na primeira iteração, o elemento lido é o `X[0]` e `i` passa a ser 2 (= 0 + 2), na segunda, o elemento lido é o `X[2]` e `i` passa a ser 4 (= 2 + 2), na terceira iteração é lido o `X[4]` e `i` passa a ser 6 (= 4 + 2), e, na quarta iteração é lido o `X[6]` e `i` passa a ser 8 (= 6 + 2). Daí, não vai mais haver a próxima iteração porque a repetição acontece somente "`enquanto (i < 7)`".

O cálculo dos valores intermediários ocorre nos passos 5 e 6. O passo 5 é a atribuição do valor inicial 1 à variável auxiliar `i`, que agora obedecerá à sequência: 1, 3, 5. A cada iteração da estrutura do passo 6, é calculado o elemento de ordem `i` e, em seguida, a variável `i` é incrementada de 2. Na primeira iteração, o elemento calculado é `X[1]=(X[1-1]+X[1+1])/2` e `i` passa ser 3 (= 1 + 2), na segunda, é calculado `X[3]=(X[3-1]+X[3+1])/2` e `i` passa ser 5 (= 3 + 2), na terceira, é calculado `X[5]=(X[5-1]+X[5+1])/2` e `i` passa ser 7 (= 5 + 2), quando pára. Finalmente, após a escrita da mensagem de saída do algoritmo no passo 7, a estrutura do passo 8 faz a escrita do vetor resultante das operações acima.

O exemplo seguinte trata de um problema clássico em computação que é a ordenação de valores armazenados em um vetor.

Exemplo 4.7

Consideremos o problema de colocar os elementos de um vetor em ordem crescente. As aplicações da solução deste problema são inúmeras. Por isso mesmo, algumas linguagens de alto nível têm o processo de ordenação predefinido entre seus comandos. Aqui, nosso

interesse é no conhecimento de um algoritmo clássico. O algoritmo a seguir é conhecido como *bubble sort* (ordenação por borbulhamento). Este método consiste em comparar os elementos vizinhos do vetor, dois a dois, e se seus valores, segundo a ordem no vetor, estiverem fora de ordem, eles são trocados de posição. Este processo é repetido até que não reste desordenado nenhum par de elementos. Tomemos um vetor `numérico vet[n]` (de tamanho `n`) o qual desejamos ordená-lo. O mapa abaixo descreve esse processo de ordenação:

```
Algoritmo
1 escrever "Digite a quantidade de elementos:"
2 ler qde
3 criar vet[qde]
4 escrever "Digite os valores:"
5 para i ← 0...qde-1, faça:
    ler vet[i]
6 trocou ← verdadeiro
7 enqanto (trocou) faça:
7.1   trocou ← falso
7.2   para i ← 0...qde-2, faça:
7.2.1       se (vet[i] > vet[i+1]) então:
7.2.1.1           aux ← vet[i]
7.2.1.2           vet[i] ← vet[i+1]
7.2.1.3           vet[i+1] ← aux
7.2.1.4           trocou ← verdadeiro
8 escrever "Vetor ordenado:"
9 para i ← 0...qde-1, faça:
      escrever vet[i]
Fim-Algoritmo
```

No algoritmo acima, os passos de 1 a 5 correspondem, respectivamente, a: leitura da quantidade `qde` de elementos, criação e leitura do vetor `vet`. Os passos 6 e 7 tratam da ordenação propriamente, e os passos 8 e 9 produzem a saída do programa que é a escrita do vetor já em ordem.

Vejamos os passos 6 e 7, onde está o centro da questão. O comando do passo 6 inicializa a variável lógica `trocou` (de nome sugestivo!) . Esta variável vai servir para indicar quando algum par de componentes do vetor for trocado (a troca somente ocorre quando elementos vizinhos estão desordenados). É inicializada com o valor `verdadeiro` para garantir a primeira execução da estrutura do passo 7. Assim que o fluxo do algoritmo entra na estrutura, no passo 7.1, a variável `trocou` recebe o valor `falso` (e somente poderá ser modificada pela estrutura de seleção 7.2.1, se houver troca). A estrutura do passo 7.2 percorre todo o vetor comparando cada elemento com o seguinte (por isso mesmo o limite da variável de controle é

uma unidade a menos do valor máximo para os índices). Toda vez que elementos vizinhos estão desordenados é feita a troca de seus conteúdos (nos passos 7.2.1.1, 7.2.1.2 e 7.2.1.3) e é modificada a variável `trocou` para `verdadeiro` (no passo 7.2.1.4), "avisando" que será preciso mais uma iteração da estrutura `enquanto`. Não haverá mais repetições somente quando a variável `trocou` se mantiver com o valor `falso` significando que não há mais vizinhos desordenados.

Quanto ao trecho do algoritmo que faz a troca de componentes do vetor, lembremos que a troca de conteúdos de variáveis já foi abordada no ***Exemplo 2.5*** (da **Subunidade 2.1.2**). Aproveitando o citado exemplo, a troca dos conteúdos de dois elementos vizinhos do vetor `vet` (`vet[i]` e seu vizinho seguinte `vet[i+1]`) foi feita usando a sequência de comandos:

(7.2.1.1) `aux` ← `vet[i]` (que salva o conteúdo de `vet[i]`)
(7.2.1.2) `vet[i]` ← `vet[i+1]` (`vet[i]` recebe `vet[i+1]`)
(7.2.1.3) `vet[i+1]` ← `aux` (`vet[i+1]` recebe o valor de `vet[i]` que foi salvo em `aux`)
onde `aux` é uma variável auxiliar.

Este algoritmo se aplica a quaisquer tipos de valores comparáveis (números, caracteres isolados ou cadeias de caracteres).

Laboratório - Vetores

Objetivos

Fixação dos conceitos relativos a vetores, incluindo criação, atribuição, leitura e escrita.

Identificação dos tipos implementados na linguagem Python para aplicação desses conceitos e testes dos algoritmos dessa subunidade.

Recursos e experimentação

Para implementar vetores, lançaremos mão do recurso de *listas* definido em Python (conforme já introduzimos no laboratório da **Subunidade 3.2.1**). Estamos cientes das propriedades da linguagem Python desde o primeiro módulo, inclusive do fato de a mesma estar fundamentada no paradigma de orientação a objetos.

Na linguagem Python, listas são objetos. Isto é, existe uma série de ações predefinidas (métodos) disponíveis para aplicação sobre esse tipo de estrutura. Sabemos também que listas podem conter dados de diferentes tipos. No entanto, estas prerrogativas não serão aqui utilizadas, considerando-se que nossas variáveis compostas serão homogêneas e que, do ponto de vista teórico, estamos adotando o paradigma estruturado.

Atribuição de valores. Recapitulando, uma lista é definida colocando-se os elementos fechados entre colchetes e separados por vírgulas. Observando-se as propriedades da linguagem, listas também não precisam ser declaradas, bastando apenas inicialização. Por

exemplo, para representar o vetor `nota` do ***Exemplo 4.1*** criamos a lista `nota`, nesse caso, inicializando com os valores dados naquele exemplo, obtemos:

```
nota = [3.0, 4.0, 8.5, 6.0]
```

Quanto aos índices, a linguagem atribui o valor zero ao índice do primeiro elemento do vetor. Logo, temos que `nota[0]` é o valor `3.0`, `nota[1]` é `4.0`, e assim por diante.

Se a intenção for alterar alguma nota isoladamente, uma atribuição direta é suficiente. Por exemplo, para substituir a nota 6.0 do vetor acima por um 7.0, basta escrever o comando `nota[3] = 7.0` (o 6,0 será automaticamente apagado). Se desejarmos criar o vetor `nota` antecipadamente (antes de atribuir valores ou fazer a leitura a partir da unidade de entrada), podemos inicializá-lo com zeros:

```
nota = [0.0, 0.0, 0.0, 0.0]
```

Esta inicialização não é prática e torna-se até inviável no caso de um vetor com uma grande quantidade de elementos. Buscando-se na linguagem Python um mecanismo para resolver este problema, encontramos o recurso de concatenação de listas. Entre outros benefícios, a concatenação permite criar dinamicamente um vetor de tamanho qualquer.

A concatenação funciona aplicando-se o operador "**+**" (como fizemos com cadeias de caracteres na **Subunidade 2.1.1**). Dadas duas listas `A` e `B`, podemos concatená-las obtendo uma nova lista `C`, usando a operação: `C = A + B`. A partir disto, podemos então afirmar que `nota = [0.0, 0.0, 0.0, 0.0]` é o resultado da seguinte concatenação:

```
nota = [0.0]+[0.0]+[0.0]+[0.0]
```

que, por sua vez, por se tratar da concatenação de listas idênticas, pode ser reduzida para:

```
nota = [0.0] * 4
```

Esta operação vai criar o vetor `nota` com quatro elementos de valor `0.0`. Este procedimento pode ser aplicado para o caso geral de inicialização de um vetor `v` de tamanho `t` escrevendo:

```
v = [0.0] * t
```

A concatenação de listas pode ainda ser aproveitada para aumentar o tamanho de um vetor, mesmo com o tamanho atribuído na inicialização. Por exemplo, para acrescentar um espaço no vetor `nota` (por exemplo, para reservar o espaço de uma futura média semestral) inicializando com zero, podemos escrever:

```
nota = nota +[0.0]
```

Ou, usando aquilo que chamamos de "atribuição composta" no laboratório da **Subunidade 2.2.1** (ver os **Experimentos 06 e 07** daquele laboratório), temos:

```
nota += [0.0]
```

(O espaço concatenado passa a fazer parte da lista) Esta propriedade pode ser bastante útil em muitas situações, inclusive o vetor pode nascer vazio (`nota = []`, por exemplo) e ser preenchido sob demanda ao longo do programa.

Leitura e escrita. Para leitura e escrita de um vetor, a linguagem Python aceita a lógica do uso de uma estrutura de repetição para acessar cada um dos componentes, como foi feito no ***Exemplo 4.3*** e no ***Exemplo 4.4***. Isto é, para ler o vetor `nota`, o código Python seria o seguinte:

```
for i in range(4):
      nota[i] = float(input())
```

E, para escrever, seria:

```
for i in range(4):
      print (nota[i])
```

Os três próximos experimentos referem-se à variável `nota` do ***Exemplo 4.1***.

Experimento 01

O programa seguinte inicializa o vetor `nota` e o escreve em seguida. Criemos o arquivo `L411_01a.py` com as seguintes linhas:

```
nota = [3.0, 4.0, 8.5, 6.0]
print ('Escrita do vetor:')
for i in range(4):
    print (nota[i])
```

Execução de `L411_01a.py`. Executando este programa, a saída será a seguinte:

```
>>>
Escrita do vetor:
3.0
4.0
8.5
6.0
>>>
```

Observamos que a saída do programa é um vetor escrito em coluna. Se quisermos escrevê-lo como um vetor em linha, uma das maneiras é manipulando a função `print()`. Esta função permite que alteremos seu padrão de escrita. Nesse caso, alteramos último caractere a ser impresso fazendo `end=' '` (espaço em branco, onde o valor padrão é "`end = '\n'` - "pula-linha"). Esta pequena alteração está no arquivo `L411_01b.py` cujas linhas de código são:

```
nota = [3.0, 4.0, 8.5, 6.0]
print ('Escrita do vetor:')
for i in range(4):
```

```
    print (nota[i],end=' ')
```

Execução de `L411_01b.py`. A saída passa a ser:

```
>>>
Escrita do vetor:
3.0 4.0 8.5 6.0
>>>
```

A linguagem Python possui mais uma facilidade que é o recurso de impressão de listas (já experimentado no laboratório da **Subunidade 3.2.1**). O programador pode utilizá-lo quando a escrita padrão for suficiente para os propósitos do programa, dispensando a estrutura de repetição (ou seja, quando não for necessário "personalizar" a escrita dos elementos do vetor). Por exemplo, para escrever o vetor `nota` na tela, digitamos:

```
print (nota)
```

Vejamos mais esta versão expressa no arquivo `L411_01c.py`:

```
nota = [3.0, 4.0, 8.5, 6.0]
print ('Escrita do vetor:')
print (nota)
```

Execução de `L411_01c.py`. A saída passa a ser:

```
>>>
Escrita do vetor:
[3.0, 4.0, 8.5, 6.0]
>>>
```

Como vimos no texto, para preencher o vetor `nota` podemos fazer atribuições diretas ou leitura do teclado. Nesse caso, o vetor já deve existir na memória do computador através da inicialização.

Experimento 02

O programa seguinte lê o vetor `nota` do teclado e o escreve na tela. Antes, cria o vetor com quatro componentes fazendo uma inicialização com zeros.

Criar o arquivo `L411_02a.py` com as seguintes linhas de código:

```
nota = [0.0]*4
print ('Vetor antes da leitura:')
print (nota)
print ('Digite os quatro elementos do vetor:')
for i in range(4):
    nota[i] = float(input())
print ('Escrita do vetor:')
```

```
print (nota)
```

O comando `nota = [0.0]*4` resulta em `nota = [0.0, 0.0, 0.0, 0.0]` que é preenchido através do teclado logo em seguida. A título de demonstração, o programa acima escreve o vetor `nota` antes e depois de sua leitura do teclado.

Execução de `L411_02a.py`. Executando este programa e digitando os valores 3.0, 4.0, 8.5 e 6.0, a saída será a seguinte:

```
>>>
Vetor antes da leitura:
[0.0, 0.0, 0.0, 0.0]
Digite os quatro elementos do vetor:
3
4
8.5
6
Escrita do vetor:
[3.0, 4.0, 8.5, 6.0]
>>>
```

O programa `L411_02b.py` a seguir produz o mesmo resultado do programa `L411_02a.py`. Todavia, nesta segunda versão, o vetor `nota` se inicia completamente vazio. Seu conteúdo vai sendo acrescido através da leitura de seus elementos via teclado.

Criar o arquivo `L411_02b.py` com o seguinte código:

```
nota = []
print ('Vetor antes da leitura:')
print (nota)
print ('Digite os quatro elementos do vetor:')
for i in range(4):
    nota += [float(input())]
print ('Escrita do vetor:')
print (nota)
```

O recurso usado continua sendo o da concatenação. Nesta ultima versão, cada novo elemento de `nota` pertence a uma lista com apenas este elemento lido do teclado.

Execução de `L411_02b.py`. Executando este programa com os mesmos dados de teste anteriores, obtemos:

```
Vetor antes da leitura:
[]
Digite os quatro elementos do vetor:
3
```

```
4
8.5
6
Escrita do vetor:
[3.0, 4.0, 8.5, 6.0]
>>>
```

Observação: As duas versões acima são igualmente úteis. Porém, a segunda versão é mais útil *quando não se sabe previamente a quantidade dos elementos do vetor*. Neste curso em particular, toda vez que o tamanho for previamente conhecido, preferiremos a inicialização com zeros.

O experimento seguinte implementa o algoritmo do ***Exemplo 4.5***.

Experimento 03

O programa abaixo lê uma dada quantidade de números reais e escreve somente aqueles que são maiores que a média aritmética entre eles.

Criar o arquivo `L411_03.py` com as seguintes linhas de código:

```
n = int(input('Digite a quantidade de números '))
vet = [0.0]*n
soma = 0.0
print ('Digite os números')
for i in range(n):
    vet[i] = float(input())
    soma += vet[i]
print ('Média aritmética:', soma/n)
print ('Valor(es) acima da média:')
for i in range(n):
    if vet[i] > soma/n: print (vet[i])
```

O primeiro comando do programa é a leitura da quantidade de elementos e o vetor é criado no comando que se segue. A criação é dinâmica. Se assim não o fosse, teríamos que fixar um tamanho máximo. A vantagem é considerável porque a reserva de memória é feita sob medida, em relação aos dados disponíveis.

Execução de `L411_03.py`. Executando este programa e digitando os valores arbitrários 45.0, 185.9, 3.3 e 266.2, a saída será a seguinte:

```
>>>
Digite a quantidade de números 4
Digite os números
45
185.9
```

```
3.3
266.2
Média aritmética: 125.1
Valor(es) acima da média:
185.9
266.2
>>>
```

Experimento 04

O ***Exemplo 4.6*** mostra um algoritmo para o preenchimento e escrita de um vetor com um tamanho fixo de sete elementos. Alguns elementos são lidos (alternadamente, e começando a leitura pelo primeiro) e os outros são calculados pela média aritmética dos seus vizinhos lidos anteriormente. Este experimento tem o objetivo de conferir o citado algoritmo.

Criar o arquivo `L411_04.py` com as seguintes linhas de código:

```
X = [0.0]*7
print ('Digite os elementos do vetor alternadamente:')
i = 0
while i < 7:
    X[i] = float(input())
    i+=2
print (X)
i = 1
while i < 6:
    X[i] = (X[i-1] + X[i+1])/2
    i+=2
print ('Vetor completo:')
print (X)
```

Execução de `L411_04.py`. Executando este programa e digitando os valores arbitrários 50.0, 100.0, 20.0 e 70.0, visualizamos:

```
>>>
Digite os elementos do vetor alternadamente:
50
100
20
70
[50.0, 0.0, 100.0, 0.0, 20.0, 0.0, 70.0]
Vetor completo:
[50.0, 75.0, 100.0, 60.0, 20.0, 45.0, 70.0]
>>>
```

O resultado confere com o esperado: 75.0 é a média aritmética entre 50.0 e 100.0, 60.0 é a média entre 100.0 e 20.0, e, 45.0 é a média entre 20.0 e 70.0.

Experimento 05

O programa do experimento anterior pode ser melhorado no sentido de valer para um tamanho qualquer do vetor `X`. O programa abaixo incrementa o citado programa, considerando agora o vetor `X` de tamanho `t` que é lido do teclado.

Criar o arquivo `L411_05.py` com as seguintes linhas de código:

```
t = int(input('Digite a quantidade de elementos (ímpar): '))
X = [0.0]*t
print ('Digite os elementos do vetor alternadamente:')
i = 0
while i < t:
    X[i] = float(input())
    i+=2
print (X)
i = 1
while i < t-1:
    X[i] = (X[i-1] + X[i+1])/2
    i+=2
print ('Vetor completo:')
print (X)
```

Nesta versão, o primeiro comando é a leitura do tamanho do vetor. Baseando-se neste valor, foram deduzidos os valores máximos para os índices representados pela variável `i`, da seguinte maneira. Na sequência de componentes a serem lidos, o limite é dado por `i < t` (ou seja, até o último índice) e na sequência dos intermediários é `i < t-1` (ou seja, até o penúltimo índice).

Execução de `L411_05.py`. Executando este programa e digitando 5 para o tamanho t do vetor e, em seguida, os valores arbitrários 30.0, 60.0 e 15.5 para as posições ímpares, visualizamos:

```
>>>
Digite a quantidade de elementos (ímpar): 5
Digite os elementos do vetor alternadamente:
30
60
15.5
[30.0, 0.0, 60.0, 0.0, 15.5]
Vetor completo:
[30.0, 45.0, 60.0, 37.75, 15.5]
```

```
>>>
```

O programa não critica o valor de `t`. Por enquanto, para seguirmos uma sequência didática de complexidade, vamos manter esta versão como está. No caso de o usuário se enganar e escrever um número par como valor de `t`, o programa não irá preencher o último componente do vetor `X`, e este elemento permanecerá com o valor zero atribuído na inicialização. *Obs.*: Caso haja interesse em aceitar apenas valores inteiros, positivos e impares para `t`, o programa deve ser alterado. A solução é simples: É bastante colocar uma estrutura de seleção para interromper o programa quando for digitado um valor inválido, ou uma de repetição se quiser fazer nova leitura de `t`.

Executando o programa e digitando, por exemplo, t = 4, e os valores 30.0 e 60.0, obtemos:

```
>>>
Digite a quantidade de elementos (ímpar): 4
Digite os elementos do vetor alternadamente:
30
60
[30.0, 0.0, 60.0, 0.0]
Vetor completo:
[30.0, 45.0, 60.0, 0.0]
>>>
```

Com tamanho `t` igual a 4, os índices pares selecionados para leitura são apenas 0 e 2 (dado que `i < t`). O índice ímpar selecionado para cálculo a partir dos seus vizinhos é apenas o 1 (dado que `i< t-1`).

Experimento 06

O programa abaixo implementa o algoritmo do ***Exemplo 4.7***. Lê um vetor de elementos numéricos e o escreve ordenado.

Criar o arquivo `L411_06.py` com as seguintes linhas de código:

```
qde = int(input('Digite a quantidade de elementos: '))
vet = [0.0]*qde
print ('Digite os valores: ')
for i in range(qde):
    vet[i] = float(input())
trocou = True
while trocou:
    trocou = False
    for i in range(qde-1):
        if vet[i] > vet[i+1]:
            aux = vet[i]
```

```
            vet[i]= vet[i+1]
            vet[i+1]= aux
            trocou = True
print ('Vetor ordenado: ')
print (vet)
```

Execução de `L411_06.py`. Executando este programa, digitando 3 para o tamanho do vetor e os números arbitrários: 355, 89.5 e 23.0, nessa ordem, visualizamos:

```
>>>
Digite a quantidade de elementos: 3
Digite os valores:
355
89.5
23
Vetor ordenado:
[23.0, 89.5, 355]
>>>
```

Observação: Convém lembrar que o trecho do programa acima que faz a troca do conteúdo entre dois componentes adjacentes do vetor pode ser reescrito da seguinte maneira (rever **Exercício de autoavaliação** da **Subunidade 2.1.3**, **item 3**):

```
        ...
        if vet[i] > vet[i+1]:
            vet[i], vet[i+1] = vet[i+1], vet[i]
            trocou = True
        ...
```

O leitor está convidado a repetir o ***Experimento 06*** modificando este trecho do programa.

Experimento 07

Neste experimento faremos uma análise do comportamento da versão do algoritmo *buble sort*, implementado no exemplo anterior. Iremos imprimir o vetor a cada iteração da estrutura `while` (ou seja, enquanto houver elementos desordenados).

Criar o arquivo `L411_07a.py` com as seguintes linhas de código:

```
qde = int(input('Digite a quantidade de elementos: '))
vet = [0.0]*qde
print ('Digite os valores: ')
for i in range(qde):
    vet[i] = float(input())
trocou = True
print ('Conferindo as iterações: ')
```

```
while trocou:
    print ('-------') # um separador das iterações
    trocou = False
    for i in range(qde-1):
        if vet[i] > vet[i+1]:
            vet[i], vet[i+1] = vet[i+1], vet[i]
            trocou = True
        print (vet)   # mostrará o vetor após cada compararação
print ('Vetor ordenado: ')
print (vet)
```

Foram acrescentadas três linhas de código: uma para anunciar a escrita, outra para escrever cada estado do vetor, e mais uma para escrever um separador de cada iteração, a fim de ajudar na análise visual. A conformação atual do vetor é escrita na tela após cada comparação entre vizinhos (tenha acontecido troca de elementos ou não) e o separador é escrito a cada iteração da estrutura `while`.

Execução de `L411_07a.py`. Executando este programa e digitando apenas os três números usados no ***Experimento 07***, visualizamos:

```
>>>
Digite a quantidade de elementos: 3
Digite os valores:
355
89.5
23
Conferindo as iterações:
-------
[89.5, 355, 23.0]
[89.5, 23.0, 355]
-------
[23.0, 89.5, 355]
[23.0, 89.5, 355]
-------
[23.0, 89.5, 355]
[23.0, 89.5, 355]
Vetor ordenado:
[23.0, 89.5, 355]
>>>
```

É interessante observar que, na primeira iteração, o maior valor (no caso, o número 355), é posicionado como último elemento do vetor. Na segunda iteração, o segundo maior valor (no caso, o número 89.5) é posicionado como penúltimo elemento do vetor. Podemos deduzir que

assim continuará para quaisquer quantidades de números. Prosseguindo, podemos ver que, a cada iteração, a necessidade de ordenação recai sobre uma menor quantidade de elementos. Nessa execução em particular, constatamos que já na segunda iteração o vetor se encontra ordenado.

A partir dessas observações, podemos concluir que o algoritmo pode ser melhorado reduzindo-se a quantidade de comparações a cada iteração da estrutura `while`. A melhoria é feita modificando-se o cabeçalho da estrutura `for`

de `"for i in range(qde-1):"`

para `"for i in range(qde-j):"`

A variável `j` armazenará um inteiro e começará valendo 0 e será incrementada de 1 a cada iteração da estrutura `while`. Ou seja, na primeira iteração os elementos serão comparados dentro da faixa de `range(qde-1)`, na segunda, dentro da faixa `range(qde-2)`, e assim por diante, como mostra o programa seguinte.

Criar o arquivo `L411_07b.py` com as seguintes linhas de código:

```
qde = int(input('Digite a quantidade de elementos: '))
vet = [0.0]*qde
print ('Digite os valores: ')
for i in range(qde):
    vet[i] = float(input())
trocou = True
print ('Conferindo as iterações: ')
j = 0
while trocou:
    j+=1  # incrementa a variável j para a próxima iteração
    print ('-------') # um separador das iterações
    trocou = False
    for i in range(qde-j):
        if vet[i] > vet[i+1]:
            vet[i], vet[i+1] = vet[i+1], vet[i]
            trocou = True
        print (vet)   # mostrará o vetor havendo troca ou não
print ('Vetor ordenado: ')
print (vet)
```

Execução de `L411_07b.py`. Executando este programa e digitando os mesmos números anteriores, obtemos:

```
>>>
Digite a quantidade de elementos: 3
```

```
Digite os valores:
355
89.5
23
Conferindo as iterações:
-------
[89.5, 355, 23.0]
[89.5, 23.0, 355]
-------
[23.0, 89.5, 355]
-------
Vetor ordenado:
[23.0, 89.5, 355]
>>>
```

Como era esperado, na primeira iteração, o maior valor (o número 355) torna-se o último elemento do vetor e, assim, restam dois números para serem ordenados. Na segunda iteração, 89.5 torna-se o penúltimo e isto encerra a ordenação porque resta apenas o número 23.0.

Exercício de autoavaliação

Realize os exercícios abaixo e discuta no fórum dos conteúdos da semana. Compare seus resultados com os dos colegas participantes. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Elabore uma nova versão do programa `L411_05.py` de modo que esta repita a leitura de `t` enquanto este não for positivo e ímpar.

2 - Construa outra versão do programa `L411_06.py` de modo que o novo objetivo seja ordenar vetores de caracteres e cadeias de caracteres. Por exemplo, executando programa para o vetor [‘Pedro’, ’Maria’, ’Ana’, ’Joaquim’], o resultado deverá ser: ['Ana', 'Joaquim', 'Maria', 'Pedro'].

3 - Qual será a conformação do vetor `nota = [3.0, 4.0, 8.5, 6.0]` após a execução do comando `nota += [0.0]`? Explique. Confirme sua resposta escrevendo o vetor antes de depois do comando dado usando o interpretador Python interativamente.

4 - Execute a sequência de comandos abaixo e, em seguida: a) Escreva um enunciado de um problema cuja solução seja o programa dado; b) Descubra e revele o que fazem as funções (assinaladas em negrito); c) Usando o conhecimento sobre estas novas funções, escreva uma versão do programa dado de modo que também seja escrita a média aritmética do números lidos.

```
v = []
print ('Digite números reais, 0-encerra:')
```

```
x = float(input('--> '))
while x!=0:
     v += [x]
     x = float(input('--> '))
print(v)
print('Foram lidos',len(v),' números dif. de zero')
if len(v)!=0:
     print('A soma dos números é',sum(v))
     print('O maior deles é',max(v))
     print('O menor é',min(v))
```

4.1.2 Matrizes

Uma matriz generaliza o vetor. Uma matriz é o caso multidimensional. Um vetor é uma matriz com apenas uma linha (ou, uma coluna). Como nos vetores, define-se uma matriz pelo seu nome e seu tamanho. Agora, o tamanho é informado relativamente a cada dimensão. Nos algoritmos, nos referiremos a uma matriz com a seguinte notação:

tipo_básico **nome_da_matriz [***tam1***] [***tam2***]... [***tamN***]**.

Onde, tam1, tam2,...,tamN referem-se às quantidades de elementos em cada dimensão. Tal como nos vetores, adotaremos índices inteiros não negativos para localizar cada elemento da matriz. Assim, os índices obedecem respectivamente às sequências: 0, 1, 2,..., *tam1-1*; 0, 1, 2,..., *tam2-1*; ... 0, 1, 2,..., *tamN-1*.

No ***Exemplo 4.1*** definimos a variável `nota` como sendo um vetor com as notas possíveis de um aluno durante um semestre letivo. São quatro notas: nota do primeiro bimestre, nota do segundo bimestre, nota de reavaliação e nota da prova final, nessa ordem. Para armazenar as notas de mais de um aluno, o exemplo seguinte mostra que o uso de uma matriz bidimensional resolve esse novo problema.

Exemplo 4.8

Consideremos que as notas possíveis de um aluno durante um semestre letivo são conforme o ***Exemplo 4.1***. e que desejamos armazenar as notas de três alunos. Podemos criar então a matriz `Nota`. Nos algoritmos, esta seria a matriz `numérica Nota[3][4]` , ou seja, a variável `Nota` é bidimensional. Em uma das dimensões, tem três linhas (uma para cada aluno) e, na outra dimensão, tem quatro colunas (uma para cada nota semestral), reservadas para conter números. Graficamente, podemos indicar a matriz `Nota` a partir de um modelo de disposição na memória como o seguinte (arbitrando-se alguns valores):

	0	1	2	3
0	3.0	4.0	8.5	6.0
1	7.5	5.5	4.5	8.0
2	6.0	7.0	5.0	7.0

O acesso a cada nota é feito através do índice respectivo. Uma referência às notas do primeiro aluno (linha de índice 0 - zero) é feita por `Nota[0][0]` para o primeiro bimestre, `Nota[0][1]` para o segundo bimestre, a reavaliação é `Nota[0][2]` e a final é `Nota[0][3]`, que valem, respectivamente, 3.0, 4.0, 8.5 e 6.0. As notas do segundo aluno (linha de índice 1) são: `Nota[1][0]` (= 7.5), `Nota[1][1]` (= 5.5), `Nota[1][2]` (= 4.5) e `Nota[1][3]` (= 8.0). As notas do terceiro aluno são: `Nota[2][0]` (= 6.0), `Nota[2][1]` (= 7.0), `Nota[2][2]` (= 5.0) e `Nota[2][3]` (= 7.0).

Atribuição de valores

Podemos preencher uma matriz atribuindo seus valores diretamente ou lendo da unidade de entrada. O cuidado com a referência individual aos elementos deve ser observado.

Exemplo 4.9

Tomemos a variável `Nota` do ***Exemplo 4.8***. Se desejarmos atribuir valores quaisquer a cada elemento da matriz, podemos fazê-lo diretamente, como os três valores abaixo:

```
Nota[1][0] ← 7.5
Nota[0][3] ← 6.0
Nota[2][1] ← 7.0
```

Os elementos de uma matriz também podem ser atribuídos já no momento de sua criação na memória do computador. As diversas linguagens de programação possuem maneiras próprias para fazer a inicialização de matrizes. Aqui, uma matriz será inicializada estendendo a notação que foi usada para vetores. Isto é, uma matriz será um "vetor" de vetores. Por exemplo, a variável `Nota` do ***Exemplo 4.8*** pode ser inicializada como:

```
Nota = [[3.0, 4.0, 8.5, 6.0], [7.5, 5.5, 4.5, 8.0], [6.0, 7.0, 5.0, 7.0]]
```

Ou seja, cada vetor que compõe `Nota` representa os dados de um aluno.

Leitura e escrita

Da mesma maneira que atribuímos valores aos elementos da matriz, podemos ler estes da unidade de entrada. Por exemplo, se quiséssemos modificar a atual nota de reavaliação (coluna de índice 2) do primeiro aluno (linha de índice 0) do ***Exemplo 4.8*** usando a leitura do teclado, seria bastante fazer:

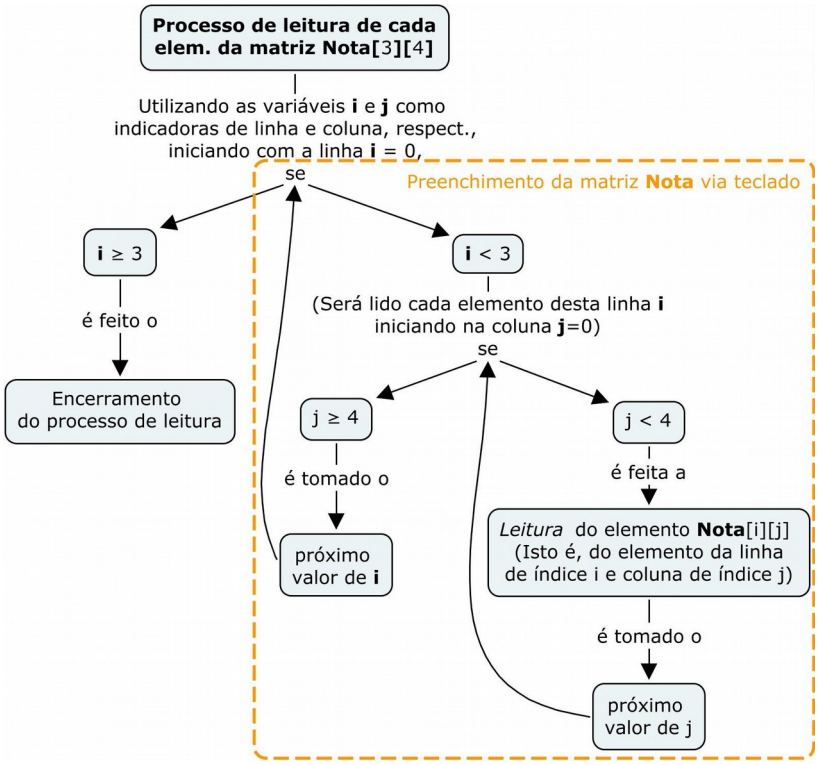
```
ler Nota[0][2]
```

Se tivéssemos que ler todas as quatro notas de todos os alunos, este comando teria que ser repetido para cada linha e para cada coluna da matriz. Portanto, a solução seria usar uma estrutura de repetição, como no exemplo seguinte.

Exemplo 4.10

Continuemos com a exploração do ***Exemplo 4.8***. O mapa a seguir demonstra como ler as notas dos três alunos, lendo cada elemento `Nota[i][j]` da linha `i` e coluna `j`, cujo trecho de algoritmo correspondente é:

```
para i ← 0...2, faça:
      para j ← 0...3, faça:
            ler Nota[i][j]
```



Ou seja, a variável `i` receberá cada índice de linha e a variável `j`, cada índice de coluna. Desse modo, para cada valor de `i` (que será 0, 1 ou 2), `j` será 0, 1, 2 ou 3. O comando `ler Nota[i][j]` será repetido para todas essas possibilidades. Isto é, na primeira iteração, o comando é `ler Nota[0][0]`, na segunda, é `ler Nota[0][1]`, na terceira, é `ler Nota[0][2]` e assim por diante.

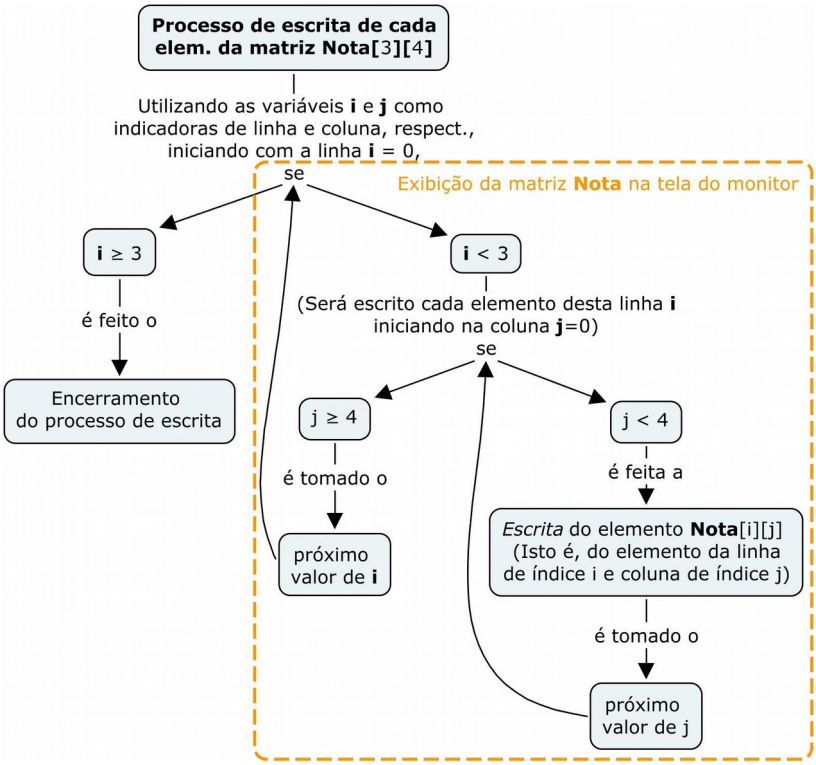
A exibição dos elementos da matriz obedece à mesma lógica aplicada na leitura. A maneira geral de escrever os componentes de uma matriz na saída é usando uma estrutura de repetição.

Exemplo 4.11

Para escrever todas as notas dos alunos do ***Exemplo 4.8***, usamos um modo análogo ao do ***Exemplo 4.10***, como mostra o trecho de algoritmo abaixo e o mapa em seguida.

Representação algorítmica da escrita da matriz `Nota`:

```
para i ← 0...2, faça:
      para j ← 0...3, faça:
            escrever Nota[i][j]
```

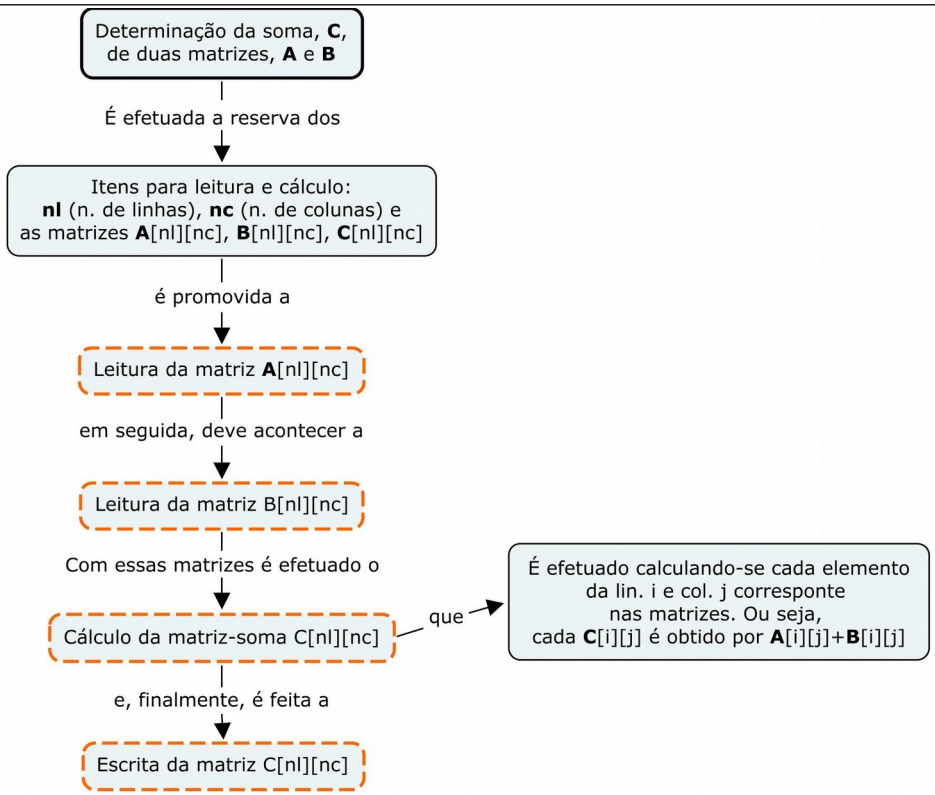


Ou seja, serão executados os comandos: `escrever Nota[0][0]`, `escrever Nota[0][1]`, `escrever Nota[0][2]`, `escrever Nota[0][3]`, `escrever Nota[1][0]`, `escrever Nota[1][1]`, `escrever Nota[1][2]`, etc.

A aplicação matemática para dessa estrutura de dados é natural. O exemplo seguinte apresenta um algoritmo para a soma de duas matrizes.

Exemplo 4.12

Vamos considerar duas matrizes bidimensionais, A e B, e que desejamos obter a matriz-soma, C, das mesmas. O algoritmo abaixo lê as matrizes A e B, juntamente com aa quantidades respectivas de linhas e colunas, e produz a matriz C. O mapa dessa solução, exibido abaixo, pode ser detalhado com facilidade tendo como base os processos de leitura e escrita de matrizes demonstrados nos exemplos ***4.10*** e ***4.11***.




```
Algoritmo
1  escrever "Número de linhas:"
2  ler nl
3  escrever "Número de colunas:"
4  ler nc
5  criar A[nl][nc]
6  criar B[nl][nc]
7  criar C[nl][nc]
8  escrever "Digite os elementos da matriz A:"
9  (Leitura da matriz A[nl][nc])
   para i ← 0...nl-1, faça:
      para j ← 0...nc-1, faça:
            ler A[i][j]
10 escrever "Digite os elementos da matriz B:"
11 (Leitura da matriz B[nl][nc])
   para i ← 0...nl-1, faça:
      para j ← 0...nc-1, faça:
            ler B[i][j]
12 (Cálculo da matriz C[nl][nc])
   para i ← 0...nl-1, faça:
      para j ← 0...nc-1, faça:
            C[i][j] ← A[i][j] + B[i][j]
```

```
13 escrever "A + B ="
14 (Escrita da matriz C[nl][nc])
   para i ← 0...nl-1, faça:
      para j ← 0...nc-1, faça:
            escrever C[i][j]
Fim-Algoritmo
```

Laboratório - Matrizes

Objetivos

Fixação dos conceitos relativos a matrizes, incluindo criação, atribuição, leitura e escrita.

Identificação dos tipos implementados na linguagem Python para aplicação desses conceitos e testes dos algoritmos dessa subunidade.

Recursos e experimentação

As matrizes, como os vetores, serão implementadas usando listas.

Atribuição de valores. No ***Exemplo 4.8*** mencionamos a variável `Nota` criada para conter as notas de três alunos. Usando os valores dados no referido exemplo, esta matriz pode ser inicializada como:

```
Nota = [[3.0, 4.0, 8.5, 6.0],[7.5, 5.5, 4.5, 8.0],[6.0, 7.0, 5.0, 7.0]]
```

Conforme esta notação, as linhas da matriz `Nota` (de índices 0, 1 e 2) são, respectivamente, os vetores: `[3.0, 4.0, 8.5, 6.0]`, `[7.5, 5.5, 4.5, 8.0]` e `[6.0, 7.0, 5.0, 7.0]`. Temos então que `Nota[0][0]` (linha 0, elemento 0) é o valor `3.0`, `Nota[0][1]` (linha 0, elemento 1) é `4.0`, `Nota[0][1]` (linha 0, elemento 2) é `8.5`, e assim por diante.

O vetor `Nota` pode ser inicializado com zeros, se pretendemos preenchê-lo com valores diferenciados somente após sua criação. Esta matriz poderia ser criada com o comando:

```
Nota = [[0.0, 0.0, 0.0, 0.0],[0.0, 0.0, 0.0, 0.0] ,[0.0, 0.0, 0.0, 0.0]]
```

Pelos motivos já expostos anteriormente, é preciso uma notação que generalize. Para cada linha dessa matriz já sabemos que a notação é `[0.0] * 4`, logo, podemos escrever:

```
Nota = [[0.0] * 4, [0.0] * 4, [0.0] * 4]
```

Todavia (**Atenção!**), a substituição por produto termina aí (se multiplicar por três linhas apenas vai criar três vetores idênticos!). A seguinte maneira de criação da matriz `Nota` é satisfatória, pois permite generalizar para qualquer quantidade de linhas e de colunas:

```
Nota = [[0] * 4 for i in range(3)]
```

Este comando vai criar a matriz `Nota` com 12 (= 3 linhas x 4 colunas) elementos de valor `0.0`. Para uma matriz `M[nlin][ncol]`, com `nlin` linhas e `ncol` colunas, a inicialização será por:

```
M = [[0] * ncol for i in range(nlin)]
```

Leitura e escrita. Para leitura e escrita de uma matriz, aplicamos a lógica mostrada no ***Exemplo 4.10*** e no ***Exemplo 4.11*** respectivamente. Isto é, para ler a matriz `Nota`, o código pode ser o seguinte:

```
for i in range(3):
      for j in range(4):
            Nota[i][j] = float(input())
```

E, para escrever:

```
for i in range(3):
      for j in range(4):
            print Nota[i][j]
```

Os três experimentos seguintes referem-se à variável `Nota` do ***Exemplo 4.8***.

Experimento 01

O programa seguinte inicializa a matriz `Nota` e a escreve em seguida. Criemos o arquivo `L412_01a.py` com as seguintes linhas de código:

```
Nota = [[3.0,4.0,8.5,6.0],[7.5,5.5,4.5,8.0],[6.0,7.0,5.0,7.0]]
print ('Escrita da matriz:')
for i in range(3):
    for j in range(4):
        print (Nota[i][j], end=' ')
```

Execução de `L412_01a.py`. Executando este programa, a saída será:

```
>>>
Escrita da matriz:
3.0 4.0 8.5 6.0 7.5 5.5 4.5 8.0 6.0 7.0 5.0 7.0
>>>
```

Observamos que o programa escreve todos os elementos da matriz na mesma linha. Podemos melhorar esta saída escrevendo cada linha da matriz em diferentes linhas também na tela do computador. Faremos isto usando o comando `print` sem argumentos, apenas para pular uma linha a cada mudança de valor da variável de controle `i`.

Criar o arquivo `L412_01b.py` com as linhas de código:

```
Nota = [[3.0,4.0,8.5,6.0],[7.5,5.5,4.5,8.0],[6.0,7.0,5.0,7.0]]
print ('Escrita da matriz:')
for i in range(3):
```

```
    for j in range(4):
        print (Nota[i][j], end=' ')
    print()  #apenas pula uma linha
```

Execução de `L412_01b.py`. A saída passa a ser a seguinte, onde as linhas estão destacadas:

```
>>>
Escrita da matriz:
3.0 4.0 8.5 6.0
7.5 5.5 4.5 8.0
6.0 7.0 5.0 7.0
>>>
```

Experimento 02

Podemos escrever a matriz `Nota` do experimento anterior em, pelo menos, mais duas formas, como mostram os programas `L412_02a.py` e `L412_02b.py` seguintes.

Criar o arquivo `L412_02a.py` com as linhas de código:

```
Nota = [[3.0,4.0,8.5,6.0],[7.5,5.5,4.5,8.0],[6.0,7.0,5.0,7.0]]
print ('Escrita da matriz:')
print (Nota)
```

A escrita da matriz sob este formato segue um padrão tal como foi definido na inicialização.

Execução de `L412_02a.py`. Executando este programa, a matriz será escrita com suas linhas indicadas entre colchetes.

```
>>>
Escrita da matriz:
[[3.0, 4.0, 8.5, 6.0], [7.5, 5.5, 4.5, 8.0], [6.0, 7.0, 5.0, 7.0]]
>>>
```

Na versão seguinte, a escrita da matriz é feita levando em conta o fato de que cada uma de suas linhas é um vetor. A matriz `Nota` possui então três linhas, de índices 0, 1 e 2, onde: a linha `[3.0, 4.0, 8.5, 6.0]` é o vetor `Nota[0]`, a linha `[7.5, 5.5, 4.5, 8.0]` é o vetor `Nota[1]`e a linha `[6.0, 7.0, 5.0, 7.0]` é o vetor `Nota[2]`.

Criar o arquivo `L412_02b.py` com as linhas de código:

```
Nota = [[3.0,4.0,8.5,6.0],[7.5,5.5,4.5,8.0],[6.0,7.0,5.0,7.0]]
print 'Escrita da matriz:'
for i in range(3):
    print (Nota[i])
```

Execução de `L412_02b.py`. Executando este programa, as linhas da matriz serão escritas em linhas distintas também na tela:

```
>>>
Escrita da matriz:
[3.0, 4.0, 8.5, 6.0]
[7.5, 5.5, 4.5, 8.0]
[6.0, 7.0, 5.0, 7.0]
>>>
```

Experimento 03

O programa seguinte lê a matriz `Nota` do teclado e a escreve na tela. Antes, a inicializa com zeros. Criar o arquivo `L412_03a.py` com as seguintes linhas de código:

```
Nota = [[0] * 4 for i in range(3)] # Isto é, ncol = 4 e nlin = 3
print ('Matriz antes da leitura:')
print (Nota)
print ('Digite os elementos por linha:')
for i in range(3):
    for j in range(4):
        Nota[i][j] = float(input())
print ('Escrita da matriz:')
print (Nota)
```

Execução de `L412_03a.py`. Executando este programa e digitando a sequência das notas de cada aluno, a saída é:

```
>>>
Matriz antes da leitura:
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
Digite os elementos por linha:
3
4
8.5
6
7.5
5.5
4.5
8
6
7
5
7
Escrita da matriz:
[[3.0, 4.0, 8.5, 6.0], [7.5, 5.5, 4.5, 8.0], [6.0, 7.0, 5.0, 7.0]]
```

```
>>>
```

Para orientar a digitação das notas, o programa seguinte incrementa a versão acima imprimindo o valor da linha a ser digitada (ver comando "`print ('linha',i)`").

Criar o arquivo `L412_03b.py` com as seguintes linhas de código:

```
Nota = [[0] * 4 for i in range(3)]
print ('Matriz antes da leitura:')
print (Nota)
print ('Digite os elementos por linha:')
for i in range(3):
    print ('linha',i)  # indica a linha a ser digitada
    for j in range(4):
        Nota[i][j] = float(input())
print ('Escrita da matriz:')
print (Nota)
```

Execução de `L412_03b.py`. Executando este programa a saída será:

```
Matriz antes da leitura:
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
Digite os elementos por linha:
linha 0
3
4
8.5
6
linha 1
7.5
5.5
4.5
8
linha 2
6
7
5
7
Escrita da matriz:
[[3.0, 4.0, 8.5, 6.0], [7.5, 5.5, 4.5, 8.0], [6.0, 7.0, 5.0, 7.0]]
>>>
```

Experimento 04

O programa abaixo lê duas matrizes bidimensionais, `A` e `B`, e produz a matriz soma, `C`. As dimensões das matrizes, `nl` e `nc`, são lidas do teclado. Trata-se da implementação do algoritmo do ***Exemplo 4.12***.

Criar o arquivo `L412_04.py` com as seguintes linhas de código:

```
nl = int(input('Número de linhas: '))
nc = int(input('Número de colunas: '))
A = [[0]*nc for i in range(nl)]
B = [[0]*nc for i in range(nl)]
C = [[0]*nc for i in range(nl)]
print ('Digite os elementos da matriz A:')
for i in range(nl):
      print ('linha',i)
      for j in range(nc):
            A[i][j] = float(input())
print ('Digite os elementos da matriz B: ')
for i in range(nl):
      print ('linha',i)
      for j in range(nc):
            B[i][j] = float(input())
for i in range(nl):
      for j in range(nc):
            C[i][j] = A[i][j] + B[i][j]
print ('A =', A)
print ('B =', B)
print ('A + B =', C)
```

Execução de `L412_04.py`. . Executando este programa entrando com as duas matrizes A e B,

$A = \begin{pmatrix} 13 & 32 \\ 67 & 15 \end{pmatrix}$ e $B = \begin{pmatrix} 57 & 22 \\ 14 & 30 \end{pmatrix}$, mostramos que a soma é $C = \begin{pmatrix} 70 & 54 \\ 81 & 45 \end{pmatrix}$

```
>>>
Número de linhas: 2
Número de colunas: 2
Digite os elementos da matriz A:
linha 0
13
32
linha 1
67
```

```
15
Digite os elementos da matriz B:
linha 0
57
22
linha 1
14
30
A = [[13.0, 32.0], [67.0, 15.0]]
B = [[57.0, 22.0], [14.0, 30.0]]
A + B = [[70.0, 54.0], [81.0, 45.0]]
>>>
```

Exercício de autoavaliação

Realize os exercícios abaixo e discuta no fórum dos conteúdos da semana. Compare seus resultados com os dos colegas participantes. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Escreva um programa para ler número real ***k***, a quantidade de linhas, a quantidade de colunas e os elementos de uma matriz bidimensional **A** (também de números reais) e determinar e escrever a matriz **B**, tal que **B** = **k***A**.

2 - Escreva um programa para ler a quantidade de linhas, a quantidade de colunas e os elementos de uma matriz bidimensional de números reais e escrever a sua *transposta*[5].

3 - São dados via teclado os elementos de ordem ímpar de uma matriz $\mathbf{A_{mXn}}$. Completar a matriz conforme a regra abaixo (tomando como exemplo a primeira sub-matriz 3x3) :

	1º.	*2º.*	*3º.*
1°	A_{11}	A_{12} = Média(A_{11}, A_{13})	A_{13}
2°	A_{21} = Média(A_{11}, A_{31})	A_{22} = Média(A_{11},A_{13},A_{31},A_3)	A_{23} = Média(A_{13}, A_{33})
3°	A_{31}	A_{32} = Média(A_{31}, A_{33})	A_{33}

4 - Consideremos os dados do ***Exemplo 4.8*** onde as notas de três alunos integram a matriz `Nota`. Considere neste exercício que a referida matriz agora armazena as notas de dez alunos. É dado também que média semestral, MS, é calculada utilizando as duas maiores notas entre avaliações bimestrais e reavaliação. Define-se a média final, MF, do aluno da seguinte maneira. Se MS for maior ou igual a 7,0 sua média final será:

MF = MS

Caso contrário, a sua média final será:

[5]Sobre *matriz transposta*, consulte a b*ibliografia complementar da disciplina no AVA Moodle*

MF = 0,6*MS + 0,4*(nota da prova final)

Faça **um** programa para:

a) Ler a matriz das notas (Obs.: Por convenção, deve ser digitado o valor zero para a reavaliação ou final que o aluno não fez);

b) Construir e escrever um vetor de dez componentes (um para cada linha da matriz `Nota`) com as respectivas médias dos alunos;

c) Determinar e escrever a maior e a menor média da turma;

d) Determinar e escrever quantos alunos foram aprovados, sabendo-se que a média para aprovação é 5,5.

(*Sugestão:* Algumas operações podem facilitadas utilizando-se as funções predefinidas citadas nos exercícios da **Subunidade 4.1.1**).

Unidade IV.2

Estruturas homogêneas especiais - Cadeias de Caracteres e Conjuntos

Nesta unidade, abordaremos duas estruturas de dados homogêneas, aqui chamadas de especiais, que podem ser construídas a partir de estruturas homogêneas básicas. Trata-se das *cadeias de caracteres* e dos *conjuntos*. As cadeias de caracteres são vistas como vetores cujos elementos são unicamente caracteres e a disposição dos elementos normalmente encerra um significado contextual. Os conjuntos são coleções em que os elementos, em princípio, estão dispostos como vetores, mas esta disposição não é requisito para distinção entre dois conjuntos. O que vale nos conjuntos é a natureza dos elementos.

4.2.1 Cadeias de Caracteres

O tipo de dado literal foi apresentado no **Modulo II** e até aprendemos a fazer operações com cadeias de caracteres na **Subunidade 2.2.3**. Nesta unidade, teremos a oportunidade de conhecer melhor a natureza deste tipo de dado, manipulável de modo semelhante aos vetores. Comumente, as linguagens de programação implementam as cadeias de caracteres com a denominação de *string*s. Por vezes, usaremos este último nome como sinônimo.

Criação e atribuição

Uma cadeia de caracteres pode ser criada a partir de diversas fontes. Uma delas é a atribuição direta. Uma variável é criada e a sequência de caracteres é retida nesta variável, tal como fizemos com os vetores. Mas, embora possamos trabalhar caractere por caractere, há normalmente o interesse em manipular a cadeia como um bloco. O contexto da *string* é sempre observado. Por exemplo, se quisermos gravar a palavra `"Maria"` na variável `nome`, fazemos apenas: `nome` ← `"Maria"`. A partir de então, como nos vetores, cada caractere da cadeia terá um índice. Assim, os elementos de `nome` são: `nome[0] (="M")`, `nome[1] (="a")`, `nome[2] (="r")`, `nome[3] (="i")` e `nome[a] (="a")`.

Leitura e escrita

A leitura e a escrita de uma string segue em geral as mesmas regras dos demais dados. Porém, a estrutura vetorial das cadeias de caracteres permite que se faça acesso a cada um dos elementos (cada caractere da cadeia). As linguagens de programação normalmente possuem uma coleção de funções predefinidas com esta finalidade. Nos algoritmos, os comandos são idênticos aos usadas para outros tipos de dados. Por exemplo, se quisermos escrever o conteúdo da variável `nome` (onde está armazenada a string `"Maria"`) escrevemos o comando: `escrever nome`. Se houver interesse em escrever apenas a inicial,

"M", por exemplo, o comando seria: `escrever nome[0]`. E, para ler um novo nome: `ler nome`.

Manipulação

Além da operação de concatenação, em muitas aplicações, há também o interesse em operações como a contagem dos elementos (determinação do tamanho da cadeia) e a extração de trechos de uma dada cadeia (extração de sub-cadeia).

Como se trata de uma estrutura vetorial, a quantidade de caracteres é um dado importante. Nesse sentido, as linguagens acrescentam algum critério (normalmente, uma marca nos próprios dados - *flag*) para que este tamanho seja determinado com facilidade através de uma função construída para esta finalidade. Se chamarmos esta função de `Tamanho()`, e dada a string nome = `"Maria"`, podemos extrair sua quantidade de elementos fazendo: `Tamanho("Maria")` ou `Tamanho(nome)`. O valor retornado nesse caso será 5.

O exemplo seguinte mostra um problema cuja solução depende do conhecimento sobre o tamanho da string.

Exemplo 4.13

Problema: Dado o nome de uma pessoa, determinar neste nome a quantidade de ocorrências da letra "A" (maiúscula ou minúscula).

```
Algoritmo
1  escrever "Digite um nome:"
2  ler nome
3  cont ← 0
4  para i ← 0...Tamanho(nome)-1, faça:
       se (nome[i]="A" ou nome[i]="a") então:
             cont ← cont + 1
5  escrever "Número de ocorrências da letra A:", cont
Fim-Algoritmo
```

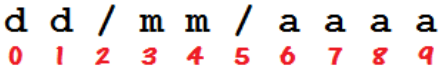
Os comandos 1 e 2 do algoritmo são destinados a leitura do nome, que é a string a ser processada. No passo 3, um contador das ocorrências da letra “A” é inicializado. O passo 4 corresponde a uma estrutura de repetição que fará uma busca pela letra “A” examinando cada caractere da cadeia. Se um caractere for igual a “A” ou a “a”, o contador `cont` é incrementado de uma unidade. Finalmente, o comando 5 faz a escrita da contagem efetuada.

Além de buscas, outra tarefa que é realizada em muitas aplicações é o recorte ou extração de sub-cadeias. O próximo exemplo apresenta um caso simples de extração de sub-cadeias para manipulação de datas.

Exemplo 4.14

Problema: É fornecida uma data no formato “dd/mm/aaaa”. Escrever apenas mês e ano no formado “mm/aaaa”.

Observamos que a data é uma cadeia com 10 caracteres e a solução do problema será a extração e escrita dos sete últimos caracteres. Ou seja, serão escritos todos caracteres a partir do índice 3:




```
Algoritmo
1  escrever "Digite uma data no formato 'dd/mm/aaaa':"
2  ler data1
3  data2 ← ''
4  para i ← 3...Tamanho(data)-1, faça:
      data2 ← data2 + data1[i]
5  escrever "Mês e ano da data fornecida: ", data2
Fim-Algoritmo
```

Os passos 1 e 2 são encarregados da leitura da data original. Mês e ano da data fornecida serão gravados na string `data2` que é inicializada no passo 3 (como string vazia). A estrutura do passo 4 concatena cada caractere extraído `data1` (a partir do índice 3 dessa string) usando a variável `data2` como destino. O último passo do algoritmo é a escrita do resultado armazenado em `data2`.

A extração de sub-cadeias, e outras operações com strings, são implementadas nas linguagens de programação através de funções predefinidas. Isto ocorre por causa da razoável frequência de aplicações. Os formatos são os mais diversos. Para extrair sub-cadeias, vamos adotar aqui, para uso em algoritmos, uma função com o seguinte formato `subcadeia(str, inicio, ncarac)`. Esta função vai extrair da string `str`, uma sub-cadeia com `ncarac`, partindo do caractere de índice `inicio`. O exemplo seguinte demonstra o uso da função ora definida.

Exemplo 4.15

O algoritmo abaixo é uma nova versão do algoritmo mostrado no exemplo anterior. Trata-se da solução do mesmo problema, mas, desta vez, o trecho associado à extração da sub-cadeia está a cargo da função `subcadeia()`.

```
Algoritmo
1  escrever "Digite uma data no formato 'dd/mm/aaaa':"
2  ler data1
3  data2 ← subcadeia(data1, 3, 7)
```

```
4  escrever "Mês e ano da data fornecida: ", data2
Fim-Algoritmo
```

Nesse algoritmo, a função `subcadeia()` é utilizada apenas uma vez. O exemplo seguinte (**Exemplo 4.16**) explora um pouco mais esta função.

Exemplo 4.16

Problema: É fornecida uma data no formato "dd/mm/aaaa". Escrever esta data no formato "aaaa/mm/dd" (formato americano).

Como vemos, a solução do problema passa por um mapeamento dos valores do dia, do mês e do ano no formato fornecido inicialmente, e estes devem ser escritos na ordem inversa. Isto é, o dia é formado por dois caracteres iniciando no índice 0 (zero), o mês corresponde a dois caracteres partindo do índice 3 e o ano deverá ser extraído a partir do índice 6 (são quatro caracteres). O algoritmo seguinte resolve o problema:

```
Algoritmo
1  escrever "Digite uma data no formato 'dd/mm/aaaa':"
2  ler data1
3  dia ← subcadeia(data1, 0, 2)
4  mes ← subcadeia(data1, 3, 2)
5  ano ← subcadeia(data1, 6, 4)
6  data2 ← ano + "/" + mes + "/" + dia
7  escrever "Data no formato americano: ", data2
Fim-Algoritmo
```

Nos dois primeiros passos do algoritmo é feita a leitura da data fornecida pelo usuário (Observação: Não discutiremos por enquanto se o usuário digitou corretamente ou não, embora já tenhamos condições de fazer esta verificação. Deixaremos este problema para o próximo módulo do curso quando poderemos utilizar recursos que tornarão a solução mais compreensiva). Os comandos 3, 4, e 5 são atribuições que extraem as sub-cadeias de interesse e o passo 6 faz a concatenação das sub-cadeias juntamente com os separadores "/", gravando o resultado em `data2`. Naturalmente, os passos 3, 4, 5, e 6 do algoritmo poderiam ser reunidos num só. No entanto, foram definidas as variáveis: dia, mes e ano com o objetivo apenas de tornar o algoritmo mais claro. O último comando, como não poderia deixar de ser, escreve o resultado.

Laboratório - Cadeias de Caracteres

Objetivos

Fixação dos conceitos básicos relativos a cadeia de caracteres e sua manipulação.

Identificação do tipo implementado na linguagem Python apropriado para aplicação desses conceitos e testes dos algoritmos dessa subunidade.

Recursos e experimentação

A linguagem Python é rica em funções predefinidas para a manipulação de cadeias de caracteres (ou, *string*s). Naturalmente, alguns problemas mostrados em nossos exemplos podem já estar resolvidos em funções embutidas na própria linguagem. Isto decorre do fato que nosso interesse está nos algoritmos. O conhecimento sobre todos os recursos da linguagem não é o mais importante nesse momento.

Em Python, uma string pode ser vista como uma lista caracteres. Vejamos o experimento seguinte.

Experimento 01

No programa a seguir, utilizaremos a função predefinida `len()` como uma implementação da função `Tamanho()` mostrada no texto. O programa determina a quantidade de ocorrências da letra "A" (maiúscula ou minúscula) em um nome fornecido pelo usuário. O algoritmo correspondente foi apresentado no ***Exemplo 4.13***.

Criar o arquivo `L421_01a.py` com as seguintes linhas:

```
nome = input('Digite um nome: ')
cont = 0
for i in range(len(nome)):
    if nome[i] in ['A','a']:
        cont += 1
    #Verif. se na posição i de nome a letra é "A" ou "a"
print ('Número de ocorrências da letra "A":', cont)
```

Observamos que a função `len()` está sendo usada para extrair a quantidade de caracteres do nome, e esta quantidade é o parâmetro da função `range()` que constrói a faixa de valores dos índices `i`.

Execução de `L421_01a.py`. Executando este programa e digitando uma string, no caso, o nome de uma pessoa, como "Maria José da Silva", por exemplo, a saída será a seguinte:

```
>>>
Digite um nome: Maria José da Silva
Número de ocorrências da letra "A": 4
>>>
```

A solução desse problema pode receber nova implementação aplicando-se a característica que associa as strings às listas de Python. Podemos então escrever a versão `L421_01b.py` abaixo. Desta vez, a variável de controle `carac` receberá seguidamente os caracteres de `nome` e assim, a comparação será direta. Ou seja, o nome é percorrido pela natureza de cada elemento e não acessando inicialmente suas posições (como foi feito no programa `L421_01a.py`).

Criar o arquivo `L421_01b.py` com usando o seguinte código:

```
nome = input('Digite um nome: ')
cont = 0
for carac in nome:
    if carac in ['A','a']:
        cont += 1
    #Verifica se cada carac de nome é "A" ou "a"
print ('Número de ocorrências da letra "A":', cont)
```

Execução de `L421_01b.py`. Executando este programa, entrando com os mesmos dados anteriores obteremos a mesma saída:

```
>>>
Digite um nome: Maria José da Silva
Número de ocorrências da letra "A": 4
>>>
```

Experimento 02

Nesse experimento implementaremos em Python a solução do problema do ***Exemplo 4.14***. Trata-se de: ler uma data no formato “dd/mm/aaaa” e escrever apenas mês e ano no formado “mm/aaaa”. O objetivo é demonstrar o algoritmo de recorte de uma cadeia de caracteres.

Criar o arquivo `L421_02.py` com as seguintes linhas:

```
data1 = input('Digite uma data no formato "dd/mm/aaaa": ')
data2 = ''
for i in range(3,len(data1)):
    data2 += data1[i]
print ('Mês e ano da data fornecida:', data2)
```

Notemos que a faixa de valores da função `range()`, "`range(3,len(data1))`", se inicia em "3" porque este é o índice do primeiro caractere relativo ao mês em `data1`. Todos os caracteres a partir desse são copiados em data2, um a um, através de concatenação.

Execução de `L421_02.py`. Executando este programa e digitando, por exemplo, "31/12/2007", a saída será a seguinte:

```
>>>
Digite uma data no formato "dd/mm/aaaa": 31/12/2007
Mês e ano da data fornecida: 12/2007
>>>
```

O próximo experimento também executará recorte de strings. Porém, desta vez, será utilizado um recurso da linguagem Python.

Experimento 03

Podemos implementar em Python a função `subcadeia(str,inicio,ncarac)` definida no texto acima, da seguinte maneira:

```
str[inicio:(inicio+ncarac)]
```

Por exemplo, se desejarmos produzir uma nova string extraindo 7 caracteres da string `data1` a partir do caractere de índice 3 escreveremos:

```
data1[3:10]
```

Esta propriedade é aplicada no programa abaixo, que é a implementação do ***Exemplo 4.15*** (lê uma data no formato “dd/mm/aaaa” e escreve apenas mês e ano no formado “mm/aaaa”).

Criar o arquivo `L421_03.py` com as seguintes linhas:

```
data1 = input('Digite uma data no formato "dd/mm/aaaa": ')
data2 = data1[3:10]
print ('Mês e ano da data fornecida:', data2)
```

Execução de `L421_03.py`. Executando este programa e digitando "31/12/2007", visualizamos:

```
>>>
Digite uma data no formato "dd/mm/aaaa": 31/12/2007
Mês e ano da data fornecida: 12/2007
>>>
```

Experimento 04

O programa abaixo, cujo algoritmo foi mostrado no ***Exemplo 4.16***, usa o recurso de substring da linguagem Python para converter uma data no formato “dd/mm/aaaa” para o formato americano, “aaaa/mm/dd”.

Criar o arquivo `L421_04.py` com as seguintes linhas:

```
data1 = input('Digite uma data no formato "dd/mm/aaaa": ')
dia = data1[0:2]  #Ou, dia = data1[:2]
mes = data1[3:5]
ano = data1[6:10] #Ou, ano = data1[6:]
data2 = ano + '/' + mes + '/' + dia
print ('Data no formato americano:', data2)
```

Consta no programa acima:

`dia = data1[0:2]`, que significa `dia = data1[0:(0+2)]` (2 caracteres a partir do 0). Também pode ser escrito: `dia = data1[0:2]`, pois, o recorte conta com o primeiro caractere.

`mes = data1[3:5]` , que significa `mes = data1[3:(3+2)]` (2 caracteres a partir do 3)

`ano = data1[6:10]` , que significa `ano = data1[6:(6+4)]` (4 caracteres a partir do 6). Também pode ser escrito: `ano = data1[6:]` pois, recorte atinge o fim da string.

Execução de `L421_04.py`. Executando este programa e digitando "31/12/2007", visualizamos:

```
>>>
Digite uma data no formato "dd/mm/aaaa": 31/12/2007
Data no formato americano: 2007/12/31
>>>
```

Exercício de autoavaliação

Realize os exercícios abaixo e discuta no fórum dos conteúdos da semana. Compare seus resultados com os dos colegas participantes. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Interativamente, atribua uma string qualquer a uma variável denominada `nome`:

```
>>> nome = '...'
>>>
```

Faça os experimentos seguintes produzindo substrings de `nome` e descreva o que acontece em cada caso:

```
>>> nome[2]
>>> nome[-2]
>>> nome[-1]
>>> nome[0:2]
>>> nome[2:0]
>>> nome[2:]
>>> nome[-2:]
>>> nome[:2]
>>> nome[0:-1]
>>> nome[1:-1]
```

Após tirar suas conclusões, crie outros exemplos e discuta com seus colegas.

2 - Escreva um programa para ler uma palavra qualquer e escrever quais vogais foram encontradas na palavra (que tenham sido acentuadas ou não). Esse processo se repete até que seja digitada uma *string* vazia. *Obs.: Para simplificar o problema, considere que sejam digitadas apenas palavras em letras maiúsculas.*

3 - Escreva uma nova versão do programa do item anterior de modo que o mesmo escreva não somente as vogais encontradas em cada palavra digitada, mas quantas unidades de cada vogal encontrada.

4.2.2 Conjuntos

Usamos *conjuntos* quando não temos interesse na posição do dado dentro da estrutura, mas apenas na sua natureza. Por exemplo, os números sorteados numa loteria podem ser armazenados numa estrutura de conjunto, pois, nesse caso, a ordem dos números é irrelevante. Nos interessamos apenas em quais números foram sorteados.

O fato da preservação da natureza dos elementos de um conjunto leva a várias possibilidades de implementação nas linguagens de programação. Inclusive, é possível que determinadas operações com conjuntos exijam a construção de estruturas até mais complexas que os vetores. O objetivo principal da abordagem desse assunto é justamente a aplicação da *Teoria dos Conjuntos*[6] na solução de problemas via computador. O objetivo aqui é apresentar apenas aspectos introdutórios.

Criação, atribuição, leitura e escrita

Limitaremos a estrutura de conjunto à representação vetorial, em princípio. Analisando em alto nível, portanto, as tarefas como criação, atribuição de valores a variáveis, leitura e escrita de conjuntos, podem ser realizadas de um modo baseado nos vetores. A diferenciação ocorrerá no modo de processamento. Ou seja, localizaremos os elementos baseando-se na indexação vetorial, mas a natureza dos elementos é que importa. Vejamos então o exemplo seguinte onde são fornecidos vetores, porém o processamento é baseado na natureza dos elementos.

Exemplo 4.17

No sorteio da Mega Sena do dia 01 de dezembro de 2007 foram sorteados os números 2, 20, 21, 27, 51 e 60. Podemos representar este resultado pelo conjunto `Sena = {2, 20, 21, 27, 51, 60}` . Dentre os milhares de jogadores, tomemos a aposta do João, que representaremos pelo conjunto `Aposta = {20, 29, 34, 44, 45, 57, 60}`. A partir desse conjunto, podemos conferir a aposta do João, e saberemos se o mesmo ganhou algum prêmio. Verificar se João foi premiado é percorrer os dois conjuntos e anotar quantos elementos do conjunto `Aposta` são também elementos do conjunto `Sena`. Podemos resolver este problema usando estruturas vetoriais para representar os conjuntos envolvidos. Nesse caso, precisamos considerar que não existem elementos repetidos em nenhum dos conjuntos. O algoritmo abaixo demonstra esta operação.

```
Algoritmo
1  Sena ← [2, 20, 21, 27, 51, 60]
2  Aposta ← [20, 29, 34, 44, 45, 57, 60]
3  cont ← 0
```

[6] *Sobre Teoria dos Conjuntos*, consulte a bibliografia complementar *da disciplina no AVA Moodle*

```
4  para i ← 0...Tamanho(Sena)-1, faça:
       para j ← 0...Tamanho(Aposta)-1, faça:
             se (Sena[i] = Aposta[j]) então:
                   cont ← cont + 1
5  escrever "Quantidade de acertos:", cont
Fim-Algoritmo
```

Nesse algoritmo destacamos o passo 4, onde é feita a contagem dos elementos comuns aos conjuntos `Sena` e `Aposta`. Cada elemento `i` do conjunto `Sena` é comparado com o elemento `j` do conjunto `Aposta`. Se forem iguais, a contagem é incrementada de uma unidade. Vemos que existem apenas dois acertos: os números 20 e 60 (Ainda não foi dessa vez que o João ficou milionário!).

Na verdade, a escolha de elementos comuns a dois conjuntos é resultado de uma operação bem conhecida em Teoria dos Conjuntos da qual trataremos logo a seguir.

Operações básicas

Todas as operações com conjuntos como estruturas de dados são oriundas da teoria que conhecemos em Matemática. Dados dois conjuntos `A` e `B`, definiremos para uso nos algoritmos: união, interseção, diferença e diferença simétrica, operações representadas com a seguinte simbologia:

União, `A ∪ B` (Conjunto dos elementos não comuns e comuns a `A` e `B`);

Interseção, `A ∩ B` (Conjunto dos elementos comuns a `A` e `B`);

Diferença, `A - B` (Conjunto dos elementos exclusivamente de `A`);

Diferença simétrica, `A Δ B` (Conjunto dos elementos que pertencem exclusivamente a `A` ou a `B`).

Outra tarefa importante é a determinação da cardinalidade (quantidade de elementos) de um conjunto. Dado um conjunto A, sua cardinalidade será representada nos algoritmos por: `card(A)`.

O ***Exemplo 4.17*** demonstra que podemos penetrar mais profundamente, explorando e construindo todas as operações com conjuntos. No entanto, as linguagens de mais alto nível já preveem estas operações e é neste caso que nos ateremos. O algoritmo a seguir envolve operações predefinidas para resolver o mesmo problema do exemplo anterior.

Exemplo 4.18

Consideremos o conjunto dos números sorteados, `Sena = {2, 20, 21, 27, 51, 60}`, e o conjunto dos números marcados pelo João, `Aposta = {20, 29, 34, 44, 45, 57, 60}`. De antemão sabemos que os acertos do João são os números que estão em `Sena` e também estão em `Aposta`. Chamemos esse conjunto de `acertos`. Ou seja, os acertos

formam a interseção entre os conjuntos dados: `acertos = Sena ∩ Aposta`. Logo, o número de acertos efetuados pelo João é extraído da cardinalidade do conjunto `acertos`, como mostra o algoritmo abaixo:

```
Algoritmo
1  Sena ← {2, 20, 21, 27, 51, 60}
2  Aposta ← {20, 29, 34, 44, 45, 57, 60}
3  acertos = Sena ∩ Aposta
4  escrever "Quantidade de acertos:", card(acertos)
Fim-Algoritmo
```

O exemplo abaixo se apresenta mais completo, pois demonstra que podemos explorar as operações com conjuntos para resolver problemas que, de outro modo, demandariam um esforço de desenvolvimento consideravelmente maior.

Exemplo 4.19

Problema: Em um certo concurso, um pequeno grupo de candidatos foi submetido a testes de Português, Matemática e Conhecimentos Gerais. Sabe-se o seguinte: Claudia, Marcel, Benito, Jorge, Francisco e Luiza são os candidatos que foram aprovados em Português; Sofia, Jorge, Marcel, Fernanda, Claudia e Francisco foram aprovados em Matemática; Claudia, Sofia, Luiza, Marcel, Alberto foram aprovados em Conhecimentos Gerais; Os candidatos Augusto e Carla foram reprovados em todas as matérias. A partir dessas informações, pergunta-se:

(a) Quais foram os candidatos que participaram do concurso?

(b) Quais candidatos foram aprovados em alguma matéria?

(c) Quais candidatos foram aprovados em todas as matérias?

(d) Quais candidatos foram aprovados em Português ou em Matemática, mas foram reprovados em Conhecimentos Gerais?

(e) Quais candidatos foram aprovados somente em Português, somente em Matemática ou somente em Conhecimentos Gerais?

Para resolver este problema, primeiramente definiremos os conjuntos de aprovados em Português, `AprPort`, em Matemática, `AprMat`, em Conhecimentos Gerais, `AprCon`, e o conjunto dos reprovados em todas as matérias, `Rep`. A partir disto, para cada questão do problema, associaremos um conjunto que responde o respectivo item. A resposta do item (a) é obtida pela determinação do conjunto universo, `U` (união de todos os conjuntos dados). A resposta do item (b) é obtida extraindo-se os reprovados do conjunto universo (será armazenada como o conjunto `A`). O item (c) é respondido determinando-se a interseção de todos os conjuntos de aprovados (será armazenada como o conjunto `B`). Para resolver o item (d) é bastante extrair os que passaram em Conhecimentos Gerais do conjunto dos aprovados em Português ou Matemática (o resultado da operação será armazenado como o conjunto `C`). A última questão corresponde a extrair do conjunto obtido no item (b), os candidatos que

passaram em duas ou três matérias. Isto equivale a extrair da diferença simétrica dos três conjuntos de aprovados, os candidatos que foram aprovados em três matérias (o resultado será armazenado como o conjunto `D`). O algoritmo abaixo resolve este problema:

```
Algoritmo
1  AprPort ← {Claudia , Marcel, Benito, Jorge, Francisco, Luiza}
2  AprMat ← {Sofia, Jorge, Marcel, Fernanda, Claudia , Francisco}
3  AprCon ← {Claudia , Sofia, Luiza , Marcel, Alberto}
4  Rep ← {Augusto, Carla}
5  U ← AprPor ∪ AprMat ∪ AprCon ∪ Rep
6  A ← U - Rep
7  B ← AprPor ∩ AprMat ∩ AprCon
8  C ← (AprPor ∪ AprMat) - AprCon
9  D ← (AprPor Δ AprMat Δ AprCon) - B
10 escrever "Todos os candidatos:",U
11 escrever "Aprovados em alguma matéria:", A
12 escrever "Aprovados em todas as matérias:", B
13 escrever "Aprov.em Port.ou em Mat., mas reprov.em conh.Gerais", C
14 escrever "Aprovados exclusivamente em Port Mat ou Conh.Gerais:", D
Fim-Algoritmo
```

Laboratório - Conjuntos

Objetivos

Fixação dos conceitos relativos ao uso da teoria dos conjuntos no computador, quanto a criação e manipulação.

Identificação do tipo implementado na linguagem Python apropriado para aplicação desses conceitos e testes dos algoritmos dessa subunidade.

Recursos e experimentação

Podemos dizer que os conjuntos em Python são listas não indexadas. Não é toda linguagem de programação que tem isto tão pronto assim. Vejamos primeiramente o experimento seguinte que mostra que é possível resolver um problema com conjuntos usando a estrutura de vetor. Logo em seguida teremos a oportunidade de usar mais recursos da linguagem Python.

Experimento 01

O programa abaixo implementa o algoritmo do ***Exemplo 4.17***. Uma aposta é representada por um vetor numérico que será comparado com outro vetor que são os números sorteados. O programa lista todos os números comuns aos dois conjuntos.

Criar o arquivo `L422_01.py` com as seguintes linhas:

```
Sena = [2, 20, 21, 27, 51, 60]
Aposta = [20, 29, 34, 44, 45, 57, 60]
cont = 0
print ('Acertos:', end=' ')
for i in range(len(Sena)):
    for j in range(len(Aposta)):
        if Sena[i] == Aposta[j]:
            print (Sena[i], end=' ')
            cont += 1
print ('\nQuantidade de acertos: ', cont)
```

Execução de `L422_01.py`. Executando este programa a saída será:

```
>>>
Acertos: 20 60
Quantidade de acertos:  2
>>>
```

Em Python, desejando representar o conjunto C = {'a', 'b', 'c'}, escrevemos como em Matemática:

```
C = {'a', 'b', 'c'}
```

Ou, convertendo uma lista para conjunto, usando a palavra `set`:

```
C = set(['a', 'b', 'c']).
```

No caso de conjunto vazio a representação é a seguinte. Por exemplo, se o conjunto `V` for inicializado como vazio, o comando correspondente será:

```
V = set()
```

Importante: Não se usa **`{}`** para conjunto vazio, pois esta representação está reservada para uma estrutura avançada de Python chamada de *dict* (ou, *dicionário*, nesse caso, dicionário vazio) que não será abordada nessa disciplina.

As operações com conjuntos também têm representação:

A União, **`A ∪ B`**, é indicada por **`A | B`**;

A Interseção, **`A ∩ B`**, é indicada por **`A & B`**;

A Diferença, **`A – B`**, é indicada por **`A - B`**;

A Diferença simétrica, **`A Δ B`**, é indicada por **`A ^ B`**.

Experimento 02

Os conjuntos de Python são utilizados no programa a seguir (implementando o algoritmo do ***Exemplo 4.18***). Os acertos são elementos da interseção entre o conjunto dos

números sorteados e a aposta, e a quantidade de acertos é a quantidade de elementos da interseção (conjunto `acertos`) extraída utilizando-se a função `len()`.

Criar o arquivo `L422_02.py` com as seguintes linhas:

```
Sena = {2, 20, 21, 27, 51, 60}
Aposta = {20, 29, 34, 44, 45, 57, 60}
acertos = Sena & Aposta
print ('Conjunto dos acertos:', acertos)
print ('Quantidade de acertos:', len(acertos))
```

Execução de `L422_02.py`. Executando este programa a saída será:

```
>>>
Conjunto dos acertos: {60, 20}
Quantidade de acertos: 2
>>>
```

Experimento 03

A solução do problema do ***Exemplo 4.19*** está implementada abaixo. O programa efetua operações com conjuntos para selecionar alunos que foram aprovados ou não nos testes de Português, Matemática e Conhecimentos Gerais. As listagens de alunos são feitas no formato de conjunto em Python.

Criar o arquivo `L422_03.py` com as seguintes linhas:

```
AprPor = {'Claudia','Marcel','Benito','Jorge','Francisco','Luiza'}
AprMat = {'Sofia','Jorge','Marcel','Fernanda','Claudia','Francisco'}
AprCon = {'Claudia','Sofia','Luiza','Marcel','Alberto'}
Rep = {'Augusto', 'Carla'}
U = AprPor | AprMat | AprCon | Rep
A = U - Rep
B = AprPor & AprMat & AprCon
C = (AprPor | AprMat) - AprCon
D = (AprPor ^ AprMat ^ AprCon) - B
print ('Todos os candidatos:',U)
print ('Aprov em alguma matéria:', A)
print ('Aprov em todas as matérias:', B)
print ('Aprov em Port ou em Mat, mas repr em Conh Gerais', C)
print ('Aprov exclusivamente em Port Mat ou Conh Gerais:', D)
```

Execução de `L422_03.py`. Executando este programa a saída será:

```
>>>
```

```
Todos os candidatos: {'Claudia', 'Augusto', 'Francisco',
'Benito', 'Sofia', 'Jorge', 'Marcel', 'Carla', 'Fernanda',
'Alberto', 'Luiza'}

Aprov em alguma matéria: {'Claudia', 'Francisco', 'Benito',
'Sofia', 'Jorge', 'Marcel', 'Fernanda', 'Alberto', 'Luiza'}

Aprov em todas as matérias: {'Claudia', 'Marcel'}

Aprov em Port ou em Mat, mas repr em Conh Gerais {'Benito',
'Francisco', 'Jorge', 'Fernanda'}

Aprov exclusivamente em Port Mat ou Conh Gerais: {'Benito',
'Fernanda', 'Alberto'}

>>>
```

Exercício de autoavaliação

Realize os exercícios abaixo e discuta no fórum dos conteúdos da semana. Compare seus resultados com os dos colegas participantes. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Atribua uma string qualquer a cada uma das variáveis denominadas `nome1` e `nome2`:

```
>>> nome1 = '...'
>>> nome2 = '...'
```

Faça logo em seguida os experimentos seguintes e descreva o que acontece em cada caso:

```
>>>
>>> set(nome1)
>>> set(nome2)
>>> set(nome1) & set(nome2)
>>> set(nome1) | set(nome2)
>>> set(nome1) ^ set(nome2)
>>> set(nome1) - set(nome2)
>>> set(nome2) - set(nome1)
```

Após tirar suas conclusões, crie outros exemplos e discuta com seus colegas.

2 - Atribua às listas `L1` e `L2` quantidades quaisquer de elementos de quaisquer tipos:

```
>>> L1 = [...]
>>> L2 = [...]
```

Em seguida, faça os experimentos abaixo e descreva o que acontece em cada caso:

```
>>>
>>> set(L1)
>>> set(L2)
>>> set(L1) & set(L2)
>>> set(L1) | set(L2)
>>> set(L1) ^ set(L2)
>>> set(L1) - set(L2)
```

```
>>> set(L2) - set(L1)
```

Tire suas conclusões, crie outros exemplos e discuta com seus colegas e tende responder: Em que casos, você considera que uso de conjuntos pode ser útil?

3 - Examine e execute código abaixo. Em seguida:

a) Elabore um enunciado para um problema cuja solução é o programa dado;

b) Comente o programa (explique o funcionamento de cada um dos seus comandos).

```
Sena = set()
Aposta = set()
print ('Digite os números sorteados na Sena: ')
while len(Sena)<6:
    Sena = Sena | {int(input('S-> '))}
print ('Digite a aposta (seis números inteiros): ')
while len(Aposta)<6:
    Aposta = Aposta | {int(input('A-> '))}
acertos = Sena & Aposta
print ('Conjunto dos acertos:', acertos)
print ('Quantidade de acertos:', len(acertos))
```

4 - Escreva uma nova versão do programa `L422_03.py` de modo que os nomes dos aprovados em Português, em Matemática, em Conhecimentos Gerais e dos reprovados em todas as matérias sejam lidos via teclado. Considere que a leitura de cada conjunto se encerra digitando-se uma string vazia, “” (isto é, uma string de tamanho igual a zero obtida digitando-se apenas a tecla <enter>).

MÓDULO V

Modularização de algoritmos

Muitas vezes, sequências de comandos reúnem características que permitem segmentar o contexto da solução de um determinado problema. Isto é, um bloco de comandos pode encerrar um sentido próprio, uma tarefa em particular.

Recordando o primeiro item do exercício de autoavaliação da **Subunidade 1.2.2**, no processo de “pagamento de uma despesa num banco”, no caso de saldo insuficiente, a opção disponível naquela questão pode ser vista aqui conceitualmente como um *módulo* (ou *subalgoritmo*): tomar "Providências para conseguir o saldo suficiente". Isto é, desde que o objetivo seja atendido, distintas providências podem ser implementadas (desejando nós que sejam todas lícitas, é claro!).

Uma das práticas mais saudáveis na solução de problemas por computador é a subdivisao em *subalgoritmos*. Vistos sob partes lógicas, problemas complexos podem ser melhor compreendidos e resolvidos.

Objetivos

- Conceituar subalgoritmos;
- Identificar subalgoritmos como segmentos da solução de problemas mais complexos;
- Permitir a construção de subalgoritmos identificando seus tipos e aplicações.

Unidades

- Unidade V.1 - Subalgoritmos
- Unidade V.2 – Recursividade

Unidade V.1

Subalgoritmos

Um subalgoritmo é um trecho de um algoritmo mais complexo cuja execução somente ocorre mediante uma ativação (ou, *chamada*) por parte do algoritmo ao qual serve. A chamada pode ser feita por outro subalgoritmo ou pelo *algoritmo principal* (aquele responsável por executar todos os processos). A solução algorítmica tem seu sentido concentrado no algoritmo principal e é por onde se inicia a execução. Os subalgoritmos são chamados pelo programa principal, ou chamam uns aos outros, seguindo seus objetivos (tantas vezes quanto for necessário).

Dentre as diversas vantagens da utilização de subalgoritmos, podemos mencionar que:

- Usando subalgoritmos, pensamos no problema em partes conceitualmente menores que, reunidas, montam a lógica da solução;

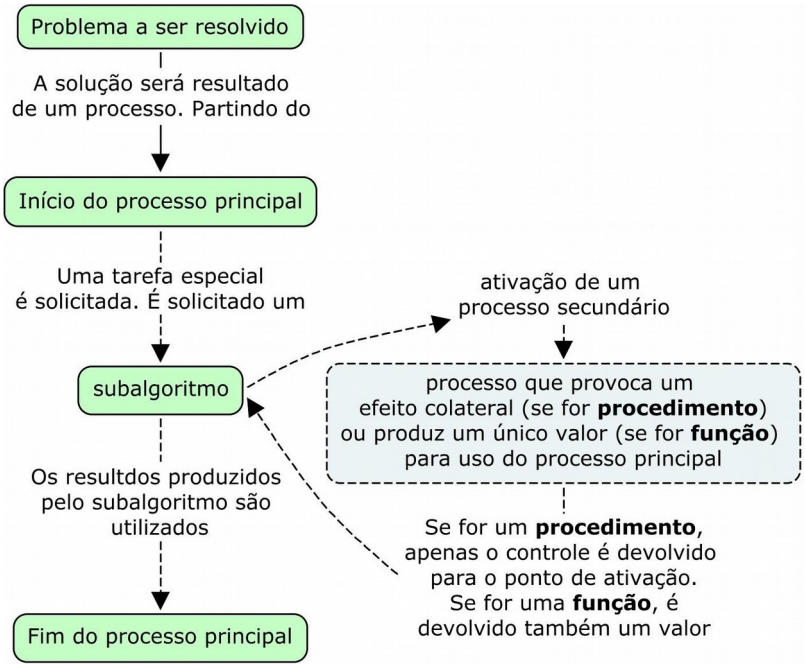
- Escreveremos menos porque repetiremos menos trechos de algoritmos; As partes facilitam a análise de erros e sua correção (um erro detectado em um subalgorimo pode ser corrigido localmente);

- Os subalgoritmos generalizam ações facilitando o reaproveitamento de código.

5.1.1 Definição - parâmetros, retorno

Os subalgoritmos realizam tarefas específicas "devolvendo", ou não, um valor para o algoritmo principal. Quando um subalgoritmo devolve um valor para o programa principal, este valor é chamado de *retorno*.

Um subalgoritmo que não retorna valor deve ser ativado de uma maneira diferente daqueles que retornam. Um subalgoritmo sem retorno é chamado na forma de um comando propriamente, e é denominado de *procedimento*. Um subalgoritmo que retorna valor é ativado ao ser inserido como parte de um comando ou expressão, e é denominado de *função*. Sob a forma de procedimento, o subalgoritmo apenas produz um determinado "efeito" como escrever, ler, modificar variáveis, etc. Na forma de função, o subalgoritmo determina um valor e o devolve diretamente ao algoritmo que o chamou. Estes conceitos estão expressos no mapa abaixo:



Os elementos processados dentro de um subalgoritmo podem ser recebidos pelo mesmo utilizando-se meios de acesso definidos especificamente com esta finalidade. Quando os dados precisarem ser comunicados desta maneira, os chamados *parâmetros* (ou, *parâmetros formais*) serão definidos. Os parâmetros são definições formais de "portas" por onde os dados são transferidos pelo algoritmo ativador. Eventualmente, estas portas também podem ser usadas para comunicar dados ao algoritmo ativador.

Para ajudar na compreensão de parâmetros e retorno, podemos mencionar as funções predefinidas já utilizadas em seções anteriores. Por exemplo, se `tan()` for uma função que calcula e retorna a tangente de um ângulo em radianos e dissermos que é ativada através da expressão `x = tan(teta)`, um número real `teta` é seu parâmetro e a variável `x` armazenará seu retorno (outro número real).

Na expressão acima citada, o algoritmo que ativa a função `tan()` insere um valor específico para `teta` que passa a se chamar de *argumento* (ou, *parâmetro real*). Isto é, argumento é o objeto de fato que atenderá ao respectivo parâmetro formal. O argumento, obviamente, deve ser aplicado de maneira compatível com o parâmetro. Isto é chamado de *passagem*. Nesse exemplo da função `tan()`, ela não tem poder de alterar o valor de `teta`. Apenas recebe uma "cópia" do seu valor para produzir um novo (a tangente). Esta associação do argumento com o parâmetro é chamada de *passagem por valor*. Muitas vezes, é conveniente que os argumentos sejam modificados pelas funções (ou procedimentos), como é o caso dos vetores e matrizes, especialmente. Nessa situação, é aplicada a *passagem como variável* (ou, *passagem por referência*). Este tipo de passagem é usado para transmitir dados ao exterior do subalgoritmo.

Com o objetivo de simplificar a linguagem, iremos generalizar a terminologia, denominando todos os subalgoritmos de *função*. A partir de agora, procedimentos ou funções

serão referidos apenas como funções. Um subalgoritmo-procedimento será uma *função sem retorno* e um subalgoritmo-função será uma *função com retorno*. Para definir uma função usaremos o seguinte formato:

```
Defina nome_da_função (parâmetros):
        corpo da função
Fim_função
```

Onde: **`nome_da_função`** é o identificador que será usado por outro algoritmo para ativar a função. Logo em seguida, entre parênteses, consta uma lista com os **`parâmetros`** da função (Os parênteses serão mantidos mesmo que a função não tenha parâmetros). O **`corpo da função`** é formado pelos comandos da função. Quando a função retornar um determinado valor, pelo menos um dos comandos deverá ser:

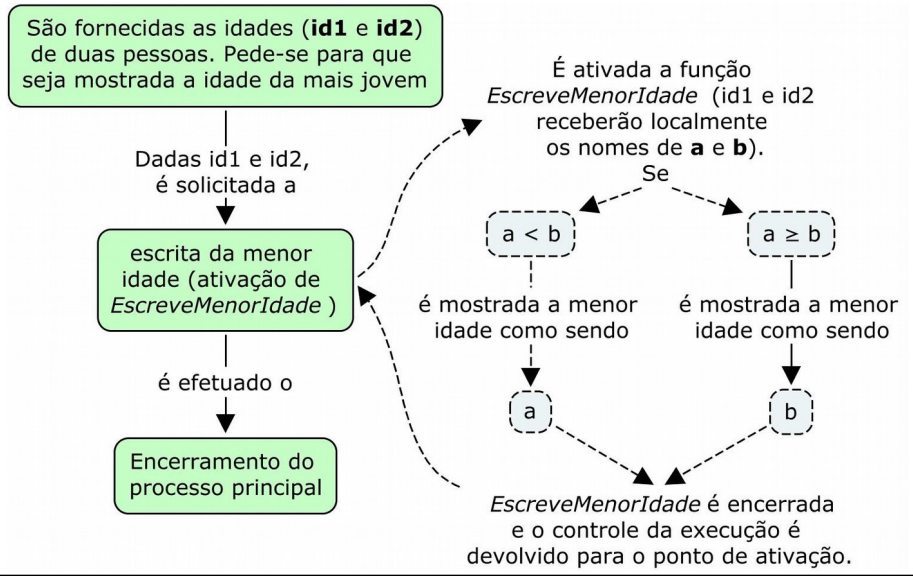
```
retornar valor
```

onde *`valor`* é o resultado produzido por comandos da função. Importante! Ao retornar o valor desejado, este comando interrompe a execução dos comandos da função em qualquer linha em que for inserido.

Exemplo 5.1

Consideremos o seguinte problema: São fornecidas as idades de duas pessoas e pede-se para que se mostre a idade da mais jovem. O problema não é tão complexo e dispensaria sua segmentação através do uso de funções. Mas, a título de ilustração, vamos explorar o uso de uma função para escolher o menor de dois valores dados. Esta função pode ser escrita com retorno ou sem retorno.

A solução abaixo considera o uso de uma função sem retorno denominada de `EscreveMenorIdade` para mostrar diretamente o menor dentre dois valores passados como argumentos. Notamos nesse caso, que a própria função se encarrega de escrever o resultado.



Em linguagem algorítmica podemos escrever:

```
Algoritmo
Defina EscreveMenorIdade(a, b):
      Se (a < b) então:
            escrever "A pessoa mais nova tem",a,"anos"
      senão:
            escrever "A pessoa mais nova tem",b,"anos"
Fim_função
1 escrever "Digite as idades de duas pessoas:"
2 ler id1
3 ler id2
4 EscreveMenorIdade(id1, id2)
Fim-Algoritmo
```

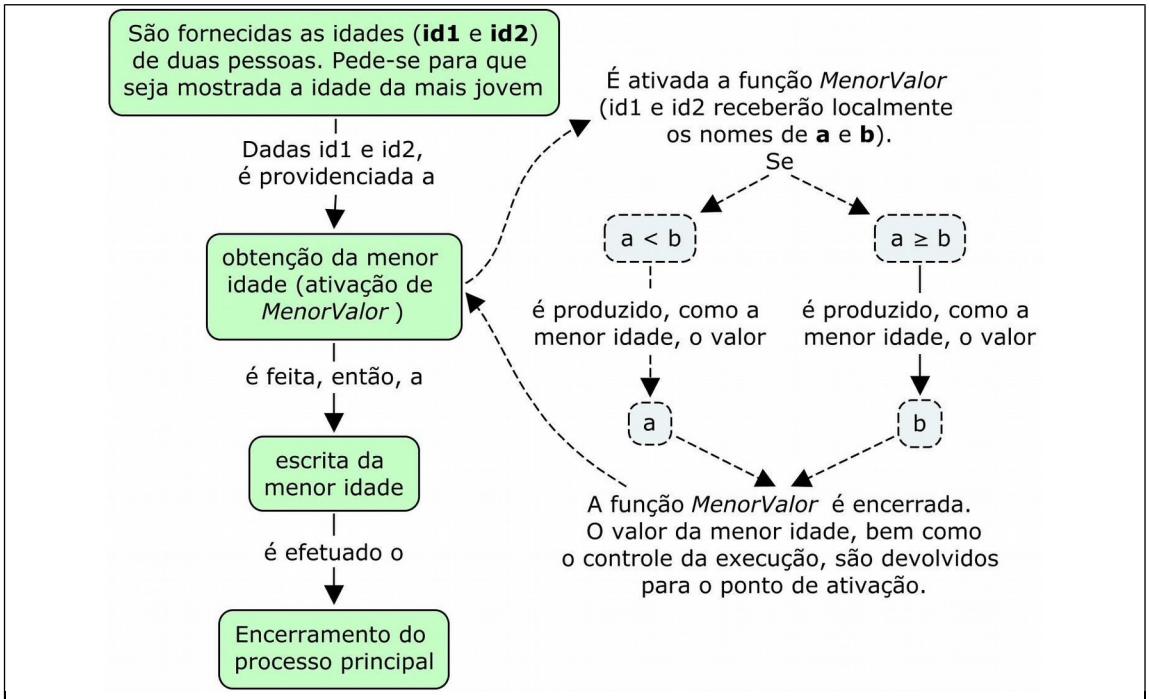
Como o algoritmo usa uma função, a primeira tarefa é a definição da mesma. A definição não faz parte dos passos do algoritmo (o algoritmo principal). Apenas a chamada da função é que está incluída entre os comandos do algoritmo principal. Nesse exemplo, é função é chamada no passo 4. Ou seja, o algoritmo principal se limita a ler as duas idades (nas variáveis `id1` e `id2`) e passar estes valores para a função `EscreveMenorIdade()`. Como havíamos exposto anteriormente, uma função sem retorno (ou procedimento) é chamada como um novo comando (observemos que o passo 4 é exclusivamente a chamada da função).

Façamos uma breve crítica a esta aplicação. Podemos observar que o corpo da função envolve elementos amarrados ao tema "idade de pessoas". Para atendender a um dos objetivos do uso de funções, devemos sempre pensar na chance de reutilização.

Vamos tomar o exemplo seguinte para demonstrar uma função reutilizável. Aproveitaremos para usar uma função com retorno.

Exemplo 5.2

Tomemos o mesmo problema do exemplo anterior. A solução a seguir usa a função denominada de `MenorValor`, que retorna o menor valor entre dois valores recebidos pela função.




```
Algoritmo
Defina MenorValor(a, b):
      Se (a < b) então:
            retornar a
      senão:
            retornar b
Fim_função
1 escrever "Digite as idades de duas pessoas:"
2 ler id1
3 ler id2
4 escrever "A pessoa mais nova tem", MenorValor(id1, id2),"anos"
Fim-Algoritmo
```

Vemos que o corpo da função inclui o comando `"retornar..."`, que faz a função ser interrompida nesse ponto, devolvendo o valor desejado para o algoritmo que a chamou. Um aspecto associado a isto é que, nesse caso em particular, a estrutura de seleção composta pode ser substituída por uma estrutura simples, mais um comando "retornar", da seguinte maneira:

```
Defina MenorValor(a, b):
      Se (a < b) então:
            retornar a
      retornar b
Fim_função
```

Esta alteração é possível pelo efeito da ação de “retornar”. Ou seja, se `a` for menor que `b`, a função retornará o valor de `a` e aí mesmo será radicalmente encerrada. Caso contrário, o próximo comando é o que será executado (nesse caso, `retornar b`).

Observações importantes podem se feitas a respeito da definição de funções. Notemos que a função apenas determina o menor valor. Quem informa que se trata de idade de pessoas é o algoritmo principal. Confere também com o exposto acima, que funções que retornam valor (funções propriamente) são chamadas como parte de comandos. No caso, a ativação de `MenorValor()` é feita inserindo-a no comando `escrever` do passo 4. Ela produz o menor valor e este ganha o sentido de “idade de pessoas” somente com a mensagem escrita.

Laboratório - Parâmetros e retorno

Objetivos

Fixação dos conceitos relativos a definição, elaboração e uso dos subalgoritmos identificando seus retornos e parâmetros.

Identificação do formato usado na linguagem Python para aplicação desses conceitos e testes dos algoritmos dessa subunidade.

Recursos e experimentação

Para implementar subalgoritmo seja no formato de procedimento ou de função propriamente, a linguagem Python aplica o título unificado de *função*, conforme introduzimos nessa subunidade. Apresenta uma sintaxe bastante simples. Inicia-se com a palavra reservada **`def`** seguida do nome da função e seus parâmetros entre parênteses, se houver (os parênteses são escritos mesmo que a função não tenha parâmetros). Logo em seguida (e endentado), após o "**`:`**" é escrito o corpo da função, da seguinte maneira:

```
def nome_da_função (parâmetros):
        corpo da função
```

As definições das funções são feitas antes de seus usos no programa. Vejamos uma função das mais simples a seguir.

Experimento 01

Experimentemos primeiramente uma definição interativa. No interpretador, escrevamos a função denominada de `avante()`. Esta função não tem parâmetros nem retorno e apenas escreve uma mensagem na tela.

```
>>> def avante():
        print ('No nosso curso:')
        print ('Vitória = Perseverança + Dedicação!')
>>>
```

Devemos observar que no uso interativo (e somente nesse caso) a função deve terminar com uma linha em branco (*prompt* surge logo em seguida). A partir de então a função está pronta para ser usada a qualquer momento. Pelo fato de a função não ter parâmetros nem retorno, sua chamada é, obviamente, digitando "`avante()`" na linha de comando. O objetivo da função será cumprido quantas vezes for chamada.

```
>>> avante()
No nosso curso:
Vitória = Perseverança + Dedicação!
>>>
```

Experimento 02

O ***Exemplo 5.1*** mostrou uma função sem retorno, porém, com parâmetros. Este experimento consta da implementação do citado algoritmo, montando-se o programa completo (com a função e o programa principal).

Criar o arquivo `L511_02.py` com as seguintes linhas de código:

```
#Função
def EscreveMenorIdade(a, b):
    if a < b:
        print ('A pessoa mais nova tem',a,'anos.')
    else:
        print ('A pessoa mais nova tem',b,'anos.')
#Programa principal
print ('Digite as idades de duas pessoas:')
id1 = int(input())
id2 = int(input())
EscreveMenorIdade(id1,id2)
```

Execução de `L511_02.py`. Executando este programa e digitando as idades de 49 e 65 anos, por exemplo, visualizamos:

```
>>>
Digite as idades de duas pessoas:
49
65
A pessoa mais nova tem 49 anos.
>>>
```

O experimento a seguir demonstra o uso de uma função com parâmetros e com retorno.

Experimento 03

O programa abaixo é a implementação do algoritmo dado no ***Exemplo 5.2***. A solução do problema inclui uma função que recebe dois valores como parâmetros e retorna o menor de

Criar o arquivo `L511_03.py` com as seguintes linhas de código:

```
#Função
def MenorValor(a, b):
    if a < b:
        return a
    return b
#Programa principal
print ('Digite as idades de duas pessoas:')
id1 = int(input())
id2 = int(input())
print ('A pessoa mais nova tem',MenorValor(id1, id2),'anos.')
```

Execução de `L511_03.py`. Executando este programa e digitando as mesmas idades usadas no experimento anterior, visualizamos exatamente o mesmo resultado:

```
>>>
Digite as idades de duas pessoas:
49
65
A pessoa mais nova tem 49 anos.
>>>
```

Exercício de autoavaliação

Resolva os exercícios abaixo com base nos conhecimentos construídos nesta subunidade. Experimente previamente e discuta no fórum dos conteúdos da semana.

1 - Elabore a função denominada de **`PesoIdeal(alt,sexo)`** que recebe a altura, **`alt`** (em metros) de uma pessoa e seu sexo, **`sexo`** (1-masculino ou 2-feminino), calcula e retorna o peso ideal p (em kg), utilizando as fórmulas empíricas: p = 72.7*altura - 58.0, se for homem e p = 62.1*altura - 44.7, se for mulher. Faça um programa que determina, usando a função elaborada, e escreve o peso ideal de uma pessoa, lendo a altura e o sexo do teclado. Obs.: Se a altura não for positiva ou o sexo não for 1 ou 2, o programa emite uma mensagem de erro.

2 - Consideremos as transações de produtos em um supermercado onde existe a seguinte relação entre códigos (inteiros de 1 a 15) e respectiva classificação dos produtos:

Código	**Classificação**
1	“Alimento não perecível”
2, 3 ou 4	“Alimento perecível”
5 ou 6	“Vestuário”
7	“Higiene pessoal”
8 até 15	“Limpeza e utensílios domésticos”

Código	Classificação
Qualquer outro código	"Código inválido"

Elaborar a função **`Classif(cod)`** que, obedecendo ao disposto no quadro acima, recebe o código, **`cod`**, como parâmetro e retorna uma string com a classificação do produto. Fazer um programa que, usando essa função, lê um código de produto e mostra sua classificação, repetindo esse processo (a leitura de códigos se encerra se o usuário digitar código 0-zero).

3 - Elabore a função **`FormamTriang(a, b, c)`**, que recebe três números inteiros, **`a`**, **`b`**, **`c`**, como parâmetros e retorna `True` se estes números puderem formar os lados de um triângulo e `False` no caso contrário. Faça um programa que lê três inteiros, verifica se eles formam triângulo usando a função elaborada. Se formar, escreve qual é o valor do maior lado.

4 - Escreva uma nova versão do programa elaborado como solução do item 3 do exercícios de autoavaliação da **Subunidade 3.1.3**, desta vez usando a função chamada **`Conceito(nota)`** que recebe a nota numérica como parâmetro e retorna o conceito ("`A`", "`B`", "`C`", "`D`" ou "`E`").

5 - Escreva uma função chamada `opera(x, y, oper)` que recebe dois números reais, `x` e `y`, e um caractere `oper` como parâmetros. Esta função retorna a soma x+y, se oper="+", a diferença x-y, se oper="-", o produto x*y, se oper="*"e o quociente x/y, se oper="/" e y for diferente de zero. Em sendo uma operação de divisão, se y for igual a zero, a função retornará `None` (*Esta é uma constante predefinida de Python que representa nulidade*). Escreva um programa simples para testar esta função (*Sugestão: Fazer um programa que lê os dois números e o sinal da operação. Em seguida, armazena o retorno da função aplicada a estes valores (por exemplo, numa variável* `x`*). Se o valor armazenado for diferente de* `None` *(isto é,* `x!=None`*) o programa escreve este valor sem problemas. Caso contrário, o programa emite uma mensagem de erro).*

5.1.2 Variáveis locais e variáveis globais

Podemos afirmar que temos até este momento uma experiência razoável com variáveis. No entanto, a novidade após a introdução das funções, é que há uma diferença entre as variáveis criadas dentro de uma função e aquelas nascidas fora dela.

Uma variável é dita *local* quando é definida dentro da função e sua existência somente é reconhecida dentro da mesma. Uma variável é denominada de *global* quando é criada fora da função. As variáveis que são globais podem ser manipuladas por quaisquer funções. Uma variável local é completamente independente de qualquer outra nascida fora da função, mesmo que tenha nomes idênticos. Dizemos que há um *escopo* para a variável. Por exemplo, as variáveis usadas na função para receber os argumentos possuem escopo local.

Exemplo 5.3

Um problema numérico clássico é o cálculo do mínimo múltiplo comum (MMC) entre números naturais. Primeiramente, lembremos que a construção dos múltiplos de um dado número natural *n* é feita através da sequência {n x 0, n x 1, n x 2, n x 3, ...}. Por exemplo, os

múltiplos de 3 correspondem à sequência {0 (= 3x0), 3 (= 3x1), 6 (= 3x2), ...} (ver ***Exemplo 3.10***). Os múltiplos de 4 correspondem à sequência {0, 4, 8, 12, ...}. Logo, os múltiplos comuns a 3 e 4 estão na sequência {0, 12, 24, ...}. O MMC de 3 e 4, por definição, será o primeiro número diferente de zero da sequência dos múltiplos comuns.

O algoritmo seguinte resolve o citado problema usando uma função que retorna o MMC de dois inteiros passados como parâmetros. Sejam x e y os parâmetros. A lógica da solução consiste em construir sequencialmente os múltiplos de `x` e cada um é verificado para ver se é tamém um múltiplo de y. Um múltiplo de `x` tem a forma `x*i`, `i`=1, 2, 3,... Este será também um múltiplo de `y` se a divisão de `x*i` por `y` for exata (com resto zero). Como a busca é em ordem crescente, o primeiro múltiplo comum encontrado será o MMC.

```
Algoritmo
Defina MMC(x, y):
(1)    i ← 1
(2)    enquanto ((x*i) mod y ≠ 0) faça:
             i = i + 1
(3)    retornar (x*i)
Fim_função
1 escrever "Digite dois números inteiros, a e b:"
2 ler a
3 ler b
4 escrever "MMC(a, b) = ", MMC(a,b)
Fim-Algoritmo
```

Foi definida a função `MMC()` e seus parâmetros são `x` e `y`. Existe ainda a variável auxiliar `i`. Estas variáveis são locais (somente podem ser referidas dentro da função). Fora da função, no algoritmo principal, estão as variáveis globais `a` e `b`. Estas variáveis são usadas para armazenar os dois números inteiros lidos do teclado e estes valores são passados para a função MMC (no passo 4). A passagem consiste numa cópia de valores.

O passo (1) da função `MMC()` é a inicialização da variável `i` cuja finalidade é construir os múltiplos de `x`; No passo (2) as repetições se seguirão até ser encontrado um múltiplo de `x` que seja também divisível por `y`. O passo (3) corresponde ao comando que faz a função retornar o primeiro múltiplo `x` que também é múltiplo de `y`. Ou seja, o menor múltiplo comum.

Podemos agora, usando funções, reescrever a solução do problema apresentado no ***Exemplo 3.15*** e avaliar quais os ganhos obtidos com a modularização.

Exemplo 5.4

Problema: Calcular e escrever o seno e o cosseno de um ângulo dado em graus, de acordo com a escolha do usuário. O algoritmo deve prever um menu com quatro opções: (1) Calcular o seno, (2) Calcular o cosseno, (3) Ler novo ângulo e (4) Encerrar.

Somente com a leitura do problema já podemos detectar seguintes ações básicas:

- Ler opção (exibindo menu)

- Calcular e escrever o seno

- Calcular e escrever o cosseno

As ações acima podem ser distribuídas, respectivamente, nas funções abaixo:

`LeOpcao()`, que exibe um menu com as quatro opções, lê e retorna a escolha do usuário;

`EscreveSeno(A)`, que recebe um ângulo `A` em graus e escreve o seno;

`EscreveCosseno(A)`, que recebe um ângulo `A` em graus e escreve o cosseno;

Devemos observar que outras ações naturalmente irão compor a solução do problema. Todavia, nem todas elas valem a pena serem transformadas em subalgoritmo. Por exemplo, a leitura de um ângulo será executada no início e quantas vezes o usuário digitar a opção 3. Como esta ação se mostra mais direta e simples, não preveremos funções para cumprir esta tarefa. Faremos uso daquelas que estiverem disponíveis na linguagem escolhida para implementação.

Considerando a existência das três funções mencionadas acima, o algoritmo da solução passa a ser o seguinte:

```
Algoritmo
Defina LeOpcao():
      escrever "  (1)Calcular o seno"
      escrever "  (2)Calcular o cosseno"
      escrever "  (3)Novo ângulo"
      escrever "  (4)Encerrar"
      escrever "Digite sua opção > "
      ler opc
      retornar opc
Fim_função
Defina EscreveSeno(A):
      escrever "O seno de",A,"graus é", seno(A*pi/180.0)
Fim_função
Defina EscreveCosseno(A):
      escrever "O cosseno de",A,"graus é",cos(A*pi/180.0)
Fim_função
1 escrever "Forneça a medida de um ângulo em graus: "
2 ler ang
3 opc ← LeOpcao()
4 enquanto(opc ≠ 4):
4.1   se (opc = 1) então: EscreveSeno(ang)
      senão se (opc = 2) então: EscreveCosseno(ang)
      senão se (opc = 3) então:
```

```
            escrever "Forneça novo ângulo: "
            ler ang
      senão: escrever "Opção inválida!"
4.2   opc ← LeOpcao()
Fim-Algoritmo
```

Comparando a versão acima com o algoritmo do ***Exemplo 3.15***, podemos notar algumas vantagens. Primeiramente, as funções tornam o texto do algoritmo mais compreensível. A saber, a função `LeOpcao()` reúne o que for necessário para a leitura da opção do usuário, diante das tarefas realizáveis pelo algoritmo. Esta função também leva à redução da quantidade de texto do algoritmo (que implica numa futura redução do código do programa) porque elimina o teste da opção 4 (de "Encerrar") da estrutura "Se" do passo 4.1. Isto acontece porque `opc` é inicializada com a primeira opção obtida do usuário. Assim o comando 4.1 é realizado com a certeza de que `opc` é diferente de 4 e a nova opção somente será conhecida no passo 4.2. As funções `EscreveSeno()` e `EscreveSeno()` reúnem as formatações de cálculo e escrita do seno e do cosseno, respectivamente. Estas funções têm apenas um comando, porém melhora a compreensão do algoritmo.

Podemos usar funções também para manipular vetores e matrizes. A leitura e a escrita são algumas tarefas mais frequentes. Uma função para leitura de um vetor, por exemplo, pode ser elaborada sob duas estratégias possíveis. A primeira é criando o vetor externamente à função e usando esta última apenas para preenchê-lo (sem retornar nenhum valor). A outra estratégia é criando e preenchendo o vetor dentro da função e retornando o vetor produzido. No exemplo seguinte escreveremos uma nova versão para do algoritmo do ***Exemplo 4.7*** usando funções.

Exemplo 5.5

Tomemos mais uma vez o problema da ordenação de um vetor. Logicamente, a solução pode ser distribuída nas seguintes tarefas:

- Ler o vetor
- Ordenar o vetor
- Escrever o vetor

Claramente, as três ações implicam em três módulos de comandos que podem dar origem a funções. Considerando que para escrever o vetor encontraremos mais facilidades para implementação, vamos elaborar somente as duas abaixo:

`LeVetor(V, t)`, que recebe o espaço de um vetor V e seu tamanho t e o preenche com dados lidos do teclado;

`OrdenaVetor(V, t)`, que recebe o vetor V de tamanho t e o ordena;

Considerando a existência as funções logo cima, o algoritmo da solução passa a ser o seguinte:

```
Algoritmo
Defina LeVetor(V, t):
      escrever "Digite os valores:"
      para i ← 0...t-1, faça:
            ler V[i]
Fim-função
Defina OrdenaVetor(V, t):
      trocou ← verdadeiro
      enqanto (trocou) faça:
            trocou ← falso
            para i ← 0...t-2, faça:
                  se (V[i] > V[i+1]) então:
                        aux ← V[i]
                        V[i] ← V[i+1]
                        V[i+1] ← aux
                        trocou ← verdadeiro
Fim-função
Algoritmo
1 escrever "Digite a quantidade de elementos:"
2 ler qde
3 criar vet[qde]
4 LeVetor(vet, qde)
5 OrdenaVetor(vet, qde)
6 escrever "Vetor ordenado:"
7 para i ← 0...qde-1, faça:
      escrever vet[i]
Fim-Algoritmo
```

A passagem de vetores (e matrizes) como parâmetros acontece por referência. Ou seja, vetores e matrizes podem ser manipulados dentro da função e a modificação é refletida exteriormente (não se tratam apenas de recepção cópias de valores). Notamos isto na função `OrdenaVetor(V, t)`. Se um componente de V é lido do teclado, este valor é preenchido no vetor passado pelo algoritmo que ativou a função. Observemos que a mesma coisa não acontece com o parâmetro t. O valor de t é passado de fora para dentro da função (é uma porta somente de entrada).

As duas funções do algoritmo acima exploram a modificação de um vetor criado externamente e passado como argumento. Já mencionamos que outra maneira de construção é criando e retornado o vetor desejado. Dado que vetores e matrizes são tratados

identicamente (variam somente em termos de dimensões), veremos a seguir um caso em que uma matriz é retornada por uma função.

Exemplo 5.6

O algoritmo abaixo é uma nova versão do algoritmo mostrado no ***Exemplo 4.12***. O problema a ser resolvido é o cálculo e escrita da soma de duas matrizes. As ações previstas

Ler as matrizes,

Calcular a soma das mesmas e

Escrever o resultado.

As matrizes são criadas no programa principal e as seguintes funções serão elaboradas para duas das ações acima:

`LeMatriz(M,l,c)`, que recebe o espaço de uma matriz `M`, o número de linhas `l` e o de colunas `c`, e a preenche com dados lidos do teclado;

`SomaMatrizes(M,N,l,c)`, que recebe as matrizes `M` e `N`, ambas com número de linhas `l` e número de colunas `c`, e retorna a matriz soma.

Considerando a existência as funções logo cima, o algoritmo da solução passa a ser o seguinte:

```
Algoritmo
Defina LeMatriz(M,l,c):
      para i ← 0...l-1, faça:
            para j ← 0...c-1, faça:
                  ler M[i][j]
Fim-função
Defina SomaMatrizes(M,N,l,c)
      criar S[l][c]
      para i ← 0...l-1, faça:
            para j ← 0...c-1, faça:
                  S[i][j] ← M[i][j] + N[i][j]
      retornar S
Fim-função
1  escrever "Número de linhas:"
2  ler nl
3  escrever "Número de colunas:"
4  ler nc
5  criar A[nl][nc]
6  criar B[nl][nc]
7  escrever "Digite os elementos da matriz A:"
8  LeMatriz(A,nl,nc)
9 escrever "Digite os elementos da matriz B:"
10 LeMatriz(B,nl,nc)
11 escrever "A + B ="
12 C = SomaMatrizes(A,B,nl,nc)
```

```
13 para i ← 0...nl-1, faça:
      para j ← 0...nc-1, faça:
            escrever C[i][j]
Fim-Algoritmo
```

No algoritmo acima todas as matrizes têm as mesmas dimensões. Em determinadas situações, surgem variáveis chaves cujos conteúdos podem ser compartilhados pelas funções, sem risco de provocar confusão até por parte do programador. Podemos, então, usar tais variáveis como globais para reduzir a quantidade de parâmetros das funções. Convém ressaltar que são situações especiais e não deve haver uso indiscriminado desse recurso.

No próximo exemplo, `nl` e `nc` são variáveis globais. Podem ser manipuladas sem que seus nomes apareçam explicitamente nas chamadas das funções.

Exemplo 5.7

O algoritmo abaixo é uma nova versão do algoritmo mostrado no ***Exemplo 5.6***. O problema a ser resolvido é o cálculo e escrita da soma de duas matrizes. Todas matrizes envolvidas têm as mesmas dimensões: nl e nc (número de linhas e de colunas respect.). Estes valores não serão acessados como argumentos das funções. Serão obtidos diretamente do algoritmo principal. As seguintes funções serão utilizadas:

`LeMatriz(M)`, que recebe o espaço de uma matriz `M` e a preenche com dados lidos do teclado;

`SomaMatrizes(M,N)`, que recebe as matrizes `M` e `N` e retorna a matriz soma.

Considerando a existência as funções logo cima, o algoritmo da solução passa a ser o seguinte:

```
Algoritmo
Defina LeMatriz(M):
      para i ← 0...nl-1, faça:
            para j ← 0...nc-1, faça:
                  ler M[i][j]
Fim-função
Defina SomaMatrizes(M,N,l,c)
      criar S[l][c]
      para i ← 0...nl-1, faça:
            para j ← 0...nc-1, faça:
                  S[i][j] ← M[i][j] + N[i][j]
      retornar S
Fim-função
1  escrever "Número de linhas:"
2  ler nl
3  escrever "Número de colunas:"
```

```
4  ler nc
5  criar A[nl][nc]
6  criar B[nl][nc]
7  escrever "Digite os elementos da matriz A:"
8  LeMatriz(A)
9 escrever "Digite os elementos da matriz B:"
10 LeMatriz(B)
11 escrever "A + B ="
12 C = SomaMatrizes(A,B)
13 para i ← 0...nl-1, faça:
      para j ← 0...nc-1, faça:
            escrever C[i][j]
Fim-Algoritmo
```

Laboratório - Variáveis locais e variáveis globais

Objetivos

Fixação dos conceitos relativos a variáveis locais e globais.

Identificação do formato usado na linguagem Python para aplicação desses conceitos e testes dos algoritmos dessa subunidade.

Recursos e experimentação

A linguagem Python implementa variável local e global tal como foi exposto acima para elaboração dos subalgoritmos. Cada variável criada dentro de uma função é variável local. Tem vida completamente independente de qualquer outra, em outra função ou no programa principal, mesmo que tenha o mesmo nome. Por outro lado, as variáveis criadas fora de todas as funções são globais e podem ser acessadas de qualquer função.

Importante: De houver interesse em alterar conteúdos de variáveis globais a partir de comandos dentro das funções, isto somente será possível se acrescentarmos a declaração:

`global` *`variável_global`*

onde *`variável_global`* é o nome da variável global que será alterada a partir de um comando da função.

Observação: Nos casos em que funções chamam outras funções, pode haver variáveis que sejam externas a uma determinada função (isto é, não sejam locais para esta função), mas, também, não sejam globais. Nessa situação, tais variáveis são declaradas em Python como **`nonlocal`**. Veremos exemplos deste uso oportunamente.

Experimento 01

Nesse experimento, usaremos a função `incrementaGlobal()` a título de demonstração. A função apenas incrementa a variável global `y`. Tomemos o arquivo `L512_01.py` com as seguintes linhas de código:

```
def incrementaGlobal():
    global y
    y += 1

# y é de escopo global (é criada a seguir fora da função)
y = int(input('Digite um número inteiro: '))
incrementaGlobal()
print ('valor incrementado:', y)
```

Execução de `L512_01.py`. Executando este programa e digitando o número 14, por exemplo, visualizamos:

```
>>>
Digite um número inteiro: 14
valor incrementado: 15
>>>
```

A declaração "`global y`" foi inserida para indicar que a variável `y` (que tem escopo global) deverá ser alterada por algum comando da função.

Importante! Sem esta declaração, `y` seria tomada como variável local. Repetindo o experimento sem esta linha de programa, uma mensagem de erro será exibida:

```
...local variable 'y' referenced before assignment
```

O erro irá ocorrer porque, localmente, haverá apenas um incremento de uma variável ainda não criada.

Experimento 02

O programa abaixo implementa o algoritmo dado no ***Exemplo 5.3***. A função envolvida retorna o MMC de dois inteiros passados como parâmetros, e usa a variável auxiliar `l`ocal i para determinar cada múltiplo do número recebido.

Criar o arquivo `L512_02.py` com as seguintes linhas de código:

```
def MMC(x, y):
    i = 1
    while (x*i)%y != 0:
        i+=1
    return x*i
print ('Digite dois números inteiros, a e b:')
```

```
a = int(input('a = '))
b = int(input('b = '))
print ('MMC(a, b) =', MMC(a,b))
```

Execução de `L512_02.py`. Executando este programa e digitando os números 45 e 60, visualizamos:

```
>>>
Digite dois números inteiros, a e b:
a = 45
b = 60
MMC(a, b) = 180
>>>
```

Experimento 03

Este experimento tem o objetivo de testar o algoritmo do ***Exemplo 5.4***. A partir de um menu, o programa oferece as opções de mostrar o seno ou o cosseno de um ângulo dado em graus, repetir o processo com um novo ângulo ou encerrar.

Criar o arquivo `L512_03.py` com as seguintes linhas de código:

```
from math import*
#Funções
def LeOpcao():
    print ('  (1)Calcular o seno')
    print ('  (2)Calcular o cosseno')
    print ('  (3)Novo ângulo')
    print ('  (4)Encerrar')
    opcao = int(input('Digite sua opção > '))
    return opcao
def EscreveSeno(A):
    print ('O seno de',A,'graus é', sin(A*pi/180.0))
def EscreveCosseno(A):
    print ('O cosseno de',A,'graus é',cos(A*pi/180.0))
#Programa principal
ang = float(input('Forneça a medida de um ângulo em graus: '))
opc = LeOpcao()
while(opc!=4):
    if   opc==1: EscreveSeno(ang)
    elif opc==2: EscreveCosseno(ang)
    elif opc==3: ang = float(input('Forneça novo ângulo: '))
    else: print ('Opção inválida!')
    opc = LeOpcao()
```

```
print ('Programa encerrado!')
```

Execução de `L512_03.py`. Executando este programa e digitando o ângulo de 60 graus e a opção 1, obtemos:

```
>>>
Forneça a medida de um ângulo em graus: 60
  (1)Calcular o seno
  (2)Calcular o cosseno
  (3)Novo ângulo
  (4)Encerrar
Digite sua opção > 1
O seno de 60 graus é 0.866025403784
  (1)Calcular o seno
  (2)Calcular o cosseno
  (3)Novo ângulo
  (4)Encerrar
Digite sua opção >
```

O programa continua ativo. Se digitarmos a seguinte sequência: opção 3 (para ler novo ângulo), 45 graus (como o novo ângulo), opção 2 (para calcular o cosseno de 45 graus) e finalmente opção 4 (para encerrar o programa), visualizamos:

```
Digite sua opção > 3
Forneça novo ângulo: 45
  (1)Calcular o seno
  (2)Calcular o cosseno
  (3)Novo ângulo
  (4)Encerrar
Digite sua opção > 2
O cosseno de 45 graus é 0.707106781187
  (1)Calcular o seno
  (2)Calcular o cosseno
  (3)Novo ângulo
  (4)Encerrar
Digite sua opção > 4
Programa encerrado!
>>>
```

Experimento 04

O programa abaixo é a implementação do ***Exemplo 5.5***. O problema resolvido é a ordenação de um vetor. São usadas duas funções: uma para ler o vetor do teclado e outra para ordená-lo.

Criar o arquivo `L512_04.py` com as seguintes linhas de código:

```
#Funções
def LeVetor(V, t):
    print ('Digite os valores: ')
    for i in range(t):
        V[i] = float(input())
def OrdenaVetor(V, t):
    trocou = True
    while trocou:
        trocou = False
        for i in range(t-1):
            if V[i] > V[i+1]:
                vet[i], vet[i+1] = vet[i+1], vet[i]
                trocou = True
#Programa principal
qde = int(input('Digite a quantidade de elementos: '))
vet = [0.0]*qde
LeVetor(vet, qde)
OrdenaVetor(vet, qde)
print ('Vetor ordenado: ')
print (vet)
```

Execução de `L512_04.py`. Executando este programa e digitando um vetor numérico com os seguintes elementos: 355, 89.5 e 23.0, por exemplo, visualizamos:

```
>>>
Digite a quantidade de elementos: 3
Digite os valores:
355
89.5
23.0
Vetor ordenado:
[23.0, 89.5, 355]
>>>
```

Experimento 05

Este experimento serve para testar o algoritmo do ***Exemplo 5.6*** que soma duas matrizes. Criar o arquivo `L512_05.py` com as seguintes linhas de código:

```
#Funções
def LeMatriz(M,l,c):
      for i in range(l):
```

```
            print ('linha',i)
            for j in range(c):
                  M[i][j] = float(input())

def SomaMatrizes(M,N,l,c):
      S = [[0]*c for i in range(l)]
      for i in range(l):
            for j in range(c):
                  S[i][j] = M[i][j] + N[i][j]
      return S
#Programa principal
nl = int(input('Número de linhas: '))
nc = int(input('Número de colunas: '))
A = [[0]*nc for i in range(nl)]
B = [[0]*nc for i in range(nl)]
print ('Digite os elementos da matriz A:')
LeMatriz(A,nl,nc)
print ('Digite os elementos da matriz B: ')
LeMatriz(B,nl,nc)
C = SomaMatrizes(A,B,nl,nc)
print ('A =', A)
print ('B =', B)
print ('A + B =', C)
```

Execução de `L512_05.py`. Executando este programa e digitando as matrizes 2 x 2

$$A = \begin{pmatrix} 18 & 5 \\ 10 & 13 \end{pmatrix} \text{ e } B = \begin{pmatrix} -6 & 10 \\ 3 & -3 \end{pmatrix}, \text{ esperamos a soma } C = \begin{pmatrix} 12 & 15 \\ 13 & 10 \end{pmatrix}.$$

Abaixo, conferimos isto:

```
>>>
Número de linhas: 2
Número de colunas: 2
Digite os elementos da matriz A:
linha 0
18
5
linha 1
10
13
Digite os elementos da matriz B:
linha 0
```

```
-6
10
linha 1
3
-3
A = [[18.0, 5.0], [10.0, 13.0]]
B = [[-6.0, 10.0], [3.0, -3.0]]
A + B = [[12.0, 15.0], [13.0, 10.0]]
>>>
```

Experimento 06

O programa abaixo também calcula e mostra a soma de duas matrizes, como no experimento anterior, porém com o uso de variáveis globais. Trata-se da implementação do algoritmo do ***Exemplo 5.7***. Mostra que as funções podem acessar variáveis globais livremente, mas não podem modificá-las. As variáveis globais são: o número de linhas e o de colunas das matrizes (pois têm os mesmos valores para todas as matrizes envolvidas no problema).

Criar o arquivo `L512_06.py` com as seguintes linhas de código:

```
#Funções
def LeMatriz(M):
    for i in range(nl):
        print ('linha',i)
        for j in range(nc):
            M[i][j] = float(input())
def SomaMatrizes(M,N):
    S = [[0]*nc for i in range(nl)]
    for i in range(nl):
        for j in range(nc):
            S[i][j] = M[i][j] + N[i][j]
    return S
#Programa principal
nl = int(input('Número de linhas: '))
nc = int(input('Número de colunas: '))
A = [[0]*nc for i in range(nl)]
B = [[0]*nc for i in range(nl)]
print ('Digite os elementos da matriz A:')
LeMatriz(A)
print ('Digite os elementos da matriz B: ')
LeMatriz(B)
C = SomaMatrizes(A,B)
print ('A =', A)
```

```
print ('B =', B)
print ('A + B =', C)
```

Execução de `L512_06.py`. Executando este programa e digitando as mesmas matrizes do experimento anterior, visualizamos um resultado também idêntico:

```
>>>
Número de linhas: 2
Número de colunas: 2
Digite os elementos da matriz A:
linha 0
18
5
linha 1
10
13
Digite os elementos da matriz B:
linha 0
-6
10
linha 1
3
-3
A = [[18.0, 5.0], [10.0, 13.0]]
B = [[-6.0, 10.0], [3.0, -3.0]]
A + B = [[12.0, 15.0], [13.0, 10.0]]
>>>
```

Exercício de autoavaliação

Resolva os exercícios abaixo com base nos conhecimentos construídos nesta subunidade. Experimente previamente e discuta no fórum dos conteúdos da semana.

1 - Escreva uma nova versão do programa `L313_01.py` (ver **Módulo III**) tal que o programa principal passe ser:

```
peso = float(input('Digite o peso(kg): '))
altura = float(input('Digite a altura(m): '))
EscreveCategoria(peso,altura)
```

Ou seja, elaborar e usar a função **`EscreveCategoria(peso,altura)`** para substituir o bloco de escrita das mensagens. Ela receberá o peso e a altura de uma pessoa como parâmetros e escreverá a categoria correspondente.

2 - Elabore uma função chamada **`Equacao2oGrau(a, b, c)`** que recebe os coeficientes **`a`**, **`b`** e **`c`** de uma equação do segundo grau ($ax^2 + bx + c = 0$, ver **Unidade III.1**) e retorna suas raízes reais como um vetor de dois componentes. Mas, se o coeficiente `a` for nulo ou a equação não tiver raízes reais, a função retornará `None`. Faça um programa para testar esta função.

3 - Elabore um programa que lê uma matriz de números reais, determina e escreve sua transposta. Elabore e use uma função para ler a matriz e outra para determinar a transposta. *Sugestão: Aproveite elementos do **Experimento 05** desta subunidade e do item 2 do exercício de autoavaliação da **Subunidade 4.1.2***.

4 - Elabore uma função que recebe um número inteiro positivo como parâmetro e retorna um vetor com seus divisores. Use essa função num programa que lê uma quantidade indeterminada de números inteiros e, a cada leitura, escreve os divisores do número lido (o programa se encerra digitando-se inteiro negativo ou zero).

Unidade V.2

Recursividade

Chama-se de *recursividade* o fato de uma função poder ser definida em termos de si mesma. Usando recursividade, problemas aparentemente complexos podem ter uma solução realmente simples e compacta. Compreender um algoritmo recursivo pode não ser trivial para o iniciante, mas é uma das formas de reduzir código de algoritmo, embora não se trate de garantia de eficiência em comparação com a função não recursiva.

5.2.1 Definição e aplicações

Tomemos a execução de uma tarefa bastante comum, como subir uma escada. Esta tarefa pode ser dada através de uma função recursiva. Denominemos esta função de `subirEscada()`, definida da maneira a seguir, onde `nd` é o número de degraus a serem vencidos. Observemos que no corpo da citada função existe uma chamada a ela mesma:

```
Defina subirEscada(nd)
      se (nd = 0) então:
            chegou!
      senão:
            subir o primeiro degrau
            subirEscada(nd-1)
Fim-função
```

A função definida desta maneira demonstra que subir uma escada é subir o primeiro degrau e o que resta após esta ação é também subir uma escada, ainda que seja uma escada menor (com um degrau a menos). A nova chamada da função acontece, de fato, como subir uma nova escada. Considerando que o processo precisa ter um fim (a escada não é infinita!), já no início da definição há um teste. O objetivo do teste é descobrir, antes de tudo, se haverá uma nova escada para subir. Cada vez que a função `subirEscada()` é chamada, o número de degraus é comparado com zero. Ora, se o número de degraus for igual a zero é porque a função não precisa mais ser chamada.

Podemos realçar pelo menos dois pontos presentes no exemplo acima, que devem servir de guia na elaboração de um algoritmo recursivo: primeiramente, uma condição de parada deve ser verificada (no exemplo, a última chamada ocorrerá quando o número de degraus for zero e isto sempre ocorrerá para se atingir o topo da escada). O outro ponto é que, a cada chamada, o espaço de trabalho vai ficando menor (no exemplo, a cada chamada o número de degraus é menor).

Exemplo 5.8

O cálculo do fatorial de um número inteiro pode ser feito através de uma função recursiva. Partimos da definição:

O fatorial de n é definido por n! = n.(n-1).(n-2)...1.

Para definição recursiva precisamos encontrar uma expressão equivalente em termos da definição de fatorial:

Sabemos que n! = n.(n - 1)!

(por exemplo, 5! = 4 . 3 . 2 . 1 equivale a 5! = 4 . 3!). Ou seja, o fatorial de um número pode ser definido em termos do fatorial dele próprio subtraído de uma unidade. Precisamos agora de uma condição de parada. Nesse caso, extraímos esta condição da própria definição. Vamos tomar o fato que 0! = 1 (também poderíamos tomar 1! = 1).

A partir dessas considerações, construímos a seguinte função recursiva:

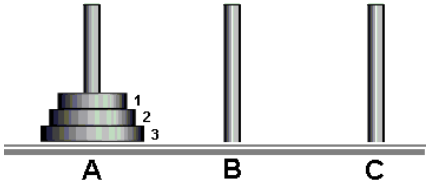
```
Defina fatorial(n):
      Se (n = 0) então:
            retornar 1
      retornar n*fatorial(n-1);
Fim-função
```

Observamos que a função `fatorial()` recebe um número inteiro positivo como parâmetro e a primeira ação é o teste do valor recebido. Decide-se dessa maneira se haverá ou não nova chamada da função recursivamente. Se n for igual a zero a função retorna 1 (internamente, isto faz disparar o desdobramento das outras chamadas, que estavam aguardando o valor definitivo). O passo seguinte dá prosseguimento chamando a função recursivamente, retornando o valor da expressão `n*fatorial(n-1)`.

O exemplo seguinte aborda o problema do jogo das "Torres de Hanói" (Ver exercícios de autoavaliação da **Unidade 1.2**). Trata-se de um caso clássico do uso de recursividade para simplificar a resolução de determinados problemas.

Exemplo 5.9

No jogo das "Torres de Hanói" existem três pinos (chamaremos de A, B e C) e n discos de diferentes tamanhos. Inicialmente, todos os discos estão no pino A.



Torre de Hanói com três discos

Sabemos que o objetivo do jogo é mover todos os discos do pino A para o C, tomando-se o pino B como temporário e que a solução deste problema é diferente para cada valor n de discos:

Se n = 1, a solução será com um único movimento:

(1) "Mover disco 1 do pino A para o pino C"

Se n = 2, a solução será:

(1) "Mover disco 1 do pino A para o pino B"
(2) "Mover disco 2 do pino A para o pino C"
(3) "Mover disco 1 do pino B para o pino C"

Pode-se mostrar que as demais soluções são deduzidas a partir destas e o resultado pode ser obtido valendo-se da recursividade, conforme função escrita a seguir:

```
Defina Hanoi(n, p1, p2, p3):
      se (n = 1)então:
            escrever "Mover disco",n,"do pino",p1,"para o pino",p3
      senão:
            Hanoi(n-1, p1, p3, p2)
            escrever "Mover disco",n,"do pino",p1,"para o pino",p3
            Hanoi(n-1, p2, p1, p3)
Fim-função
```

Laboratório - Definição e aplicações

Objetivos

Fixar os conceitos relativos a definição de funções recursivas

Identificação do formato usado na linguagem Python para aplicação desses conceitos e testes dos algoritmos dessa subunidade.

Recursos e experimentação

A linguagem Python permite a implementação de funções recursivas, como iremos conferir nos experimentos seguintes.

Experimento 01

Este experimento implementa a função recursiva do ***Exemplo 5.8*** para escrever o fatorial de um numero lido do teclado. O numero deve ser inteiro e positivo. Caso contrário será emitida uma mensagem de erro.

Criar o arquivo `L521_01.py` com as seguintes linhas de código:

```
def fatorial(n):
    if n == 0:
        return 1
    return n*fatorial(n-1)

n = int(input('Digite um número inteiro positivo: '))
if (n >= 0):
    print ('O fatorial de',n,'é',fatorial(n))
else:
    print ('Dado inválido!')
```

Execução de `L521_01.py`. Testando este programa com os números 0, 1, 2 e 5 seguidamente, encontramos os resultados:

```
>>>
Digite um número inteiro positivo: 0
O fatorial de 0 é 1
>>> ========================= RESTART ============================
>>>
Digite um número inteiro positivo: 1
O fatorial de 1 é 1
>>> ========================= RESTART ============================
>>>
Digite um número inteiro positivo: 2
O fatorial de 2 é 2
>>> ========================= RESTART ============================
>>>
Digite um número inteiro positivo: 5
O fatorial de 5 é 120
>>>
```

Experimento 02

O ***Exemplo 5.9*** mostrou uma função recursiva para resolver o problema do jogo das Torres de Hanói. O programa a seguir lê a quantidade de discos em jogo e chama esta função para listar os passos que resolvem o problema. O programa passa para a função a quantidade de discos e a denominação (A, B e C) dos pinos.

Criar o arquivo `L521_02.py` com as seguintes linhas de código:

```
def Hanoi(n, p1, p2, p3):
    if n==1:
        print('Mover disco',n,'do pino',p1,'para o pino',p3)
    else:
        Hanoi(n-1, p1, p3, p2)
        print ('Mover disco',n,'do pino',p1,'para o pino',p3)
        Hanoi(n-1, p2, p1, p3)
#Programa principal
n = int(input('Quantidade de discos: '))
if n > 0:
    print ('Solução:' )
    Hanoi(n,'A','B','C')
else:
    print ('Dado inválido!')
```

Execução de `L521_02.py`. Executando este programa para 1, 2 e 3 discos, obtemos o seguinte resultado:

```
>>>
Quantidade de discos: 1
Solução:
Mover disco 1 do pino A para o pino C
>>> ========================= RESTART ============================
>>>
Quantidade de discos: 2
Solução:
Mover disco 1 do pino A para o pino B
Mover disco 2 do pino A para o pino C
Mover disco 1 do pino B para o pino C
>>> ========================= RESTART ============================
>>>
Quantidade de discos: 3
Solução:
Mover disco 1 do pino A para o pino C
Mover disco 2 do pino A para o pino B
Mover disco 1 do pino C para o pino B
Mover disco 3 do pino A para o pino C
Mover disco 1 do pino B para o pino A
Mover disco 2 do pino B para o pino C
Mover disco 1 do pino A para o pino C
>>>
```

Exercício de autoavaliação

Resolva os exercícios abaixo com base nos conhecimentos construídos nesta subunidade. Experimente previamente e discuta no fórum dos conteúdos da semana.

1 - Implemente a função recursiva `pot(x,y)`, que recebe dois números reais `x` e `y` como parâmetros e retorna o valor de x^y. Escreva um programa que aplique esta função *(Sugestão: Ver que* $x^y = x*x^{(y-1)}$ *e* $x^0 = 1$*. Por exemplo,* $3^5 = 3*3^4$, $3^4 = 3*3^3$, *...)*.

2 - Faça um programa para experimentar e descobrir o que a função abaixo escreve:

```
def Escreve (num):
    if num>0:
        print(num)
        Escreve(num-1)
```

Mostre o que muda no resultado se a função for reescrita assim:

```
def Escreve2(num):
    if num>0:
        Escreve2(num-1)
        print(num)
```

(Isto é, verificar o que mudará quando a ordem dos dois últimos comandos for invertida). Tente explicar a diferença entre os resultados produzidos.

3 - Elabore uma função recursiva que apenas escreve em coluna os elementos de um dado vetor passado como parâmetro. Faça um programa simples para demonstrar o uso da função elaborada. *Sugestão: Escrever os elementos de um vetor recursivamete implementa a ideia da função* `subirEscada(nd)` *apresentada na introdução da* ***Subunidade 5.2.1****. Isto é, no corpo da função haverá a escrita do primeiro elemento do vetor, seguido de uma chamada recursiva da função tomando-se o vetor a partir do segundo elemento.*