Laboratório de Simulação Matemática

Notas de Aula

Prof. Donald Mark Santee

Índice

Introdução

O uso do computador amplia muito a capacidade do matemático, engenheiro, cientista e matemático industrial de resolver, prever e analisar problemas e situações. Entretanto não basta ter um computador na mão, é necessário saber o que fazer com ele e, saber das suas limitações.

Para muitos tipos de problemas já existem aplicativos prontos, inclusive que podem ser usados via internet ou até via celular, que irão auxiliar o profissional no seu trabalho. Contudo para problemas mais novos ou mais complexos essas facilidades podem não existir, e a capacidade de trabalhar esses problemas mais complexos é que vai justificar ao profissional o seu título de matemático, matemático industrial, engenheiro ou cientista.

Para os problemas e situações “fora do padrão” é necessário que se ensine o computador, com detalhes, o que ele deve fazer. A esse processo de instruir passo a passo o que o computador deve fazer chama-se de *programação* do computador.

Existem vários tipos de situações em que é necessário programar um computador: administração do acesso a bancos de dados, comunicação com a internet, edição de imagens, automação industrial etc. Dentre estes, os tipos de problemas, os que são enfocados aqui são os chamados *Métodos Numéricos*, que são receitas para a solução de determinados tipos de problemas muito abrangentes e comuns. Além do estudo dos métodos numéricos, questões sobre vantagens e desvantagens de diversas maneiras de resolver um mesmo problema serão discutidos.

Para instruir o computador a buscar uma solução para um determinado problema, é necessário que se use uma linguagem que ambos, programador e computador, entendam. A linguagem interna de um computador, chamada *linguagem de máquina*, decorrente dos seus circuitos digitais, é uma linguagem binária, normalmente representada pelos humanos por zeros e uns. Dar instruções a um computador nessa linguagem é tedioso e improdutivo, além disso os comandos teriam que ser adaptados para cada máquina que se desejasse programar. Para contornar esses problemas, foram desenvolvidas linguagens próprias para se comunicar com o computador, chamadas de *linguagens de programação*.

Numa linguagem de programação cria-se um texto em uma linguagem intermediária clara e objetiva, e delega-se a conversão para a linguagem de máquina a um computador, que poderá até ser o próprio computador que irá executar essas instruções. Uma linguagem de programação é, portanto, uma linguagem artificial projetada para dar instruções para uma máquina. Ela é usada para criar programas que controlam o comportamento da máquina. Esses programas são comumente chamados de *código fonte*.

As primeiras linguagens de programação são anteriores à invenção do computador, e foram usadas para configurar o comportamento de máquinas como o Tear de Jacquard e os Pianos Automáticos. Existem, e foram criadas ao longo da história, centenas, e talvez até milhares de linguagens de programação, variando desde linguagens dependentes de uma máquina específica até linguagens de uso geral. A tabela abaixo lista algumas que merecem destaque:

Linguagem	Ano	Observação
ASSEMBLY	Anterior a 1950	As instruções do processador são codificadas diretamente através de mnemônicos. É uma linguagem totalmente

		dependente do tipo do processador e de seus periféricos. Apesar disso é utilizada até hoje na indústria programação de dispositivos e interfaces.
FORTRAN (*FORmula TRAnslation*)	1950	Desenvolvida para aplicações científicas. Foi a linguagem predominante no meio científico por mais de trinta anos. Até hoje é a linguagem usada em alguns centros de pesquisa e empresas de desenvolvimento.
BASIC (*Begginers All-purpose Symbolic Instruction Code*)	1964	Primeira linguagem criada com fins didáticos para ser usada no ensino de programação. É a base do *Visual-Basic* utilizado nas linguagens de macro dos pacotes *office* atuais.
C	1970	Linguagem criada nos Laboratórios Bell. Suas sucessoras C++ e C# são as mais utilizadas hoje.
PASCAL	1970	Criada para ser uma linguagem de fácil aprendizado, e que estimulasse os bons hábitos de programação, é a base dos ambientes de programação *Delphi*, *Kylix* e *Lazarus* para desenvolvimento de aplicativos com excelente interface com o usuário.
Java	1991	Com a sintaxe baseada na linguagem C o java inova na criação do conceito de Máquina Virtual. Que torna os aplicativos independentes de plataforma.

A linguagem adotada para programação neste texto será o *Python*. Supõe-se que o leitor não tem conhecimento de nenhuma outra linguagem, assim não serão feitas comparações entre linguagens, e cada novo conceito de programação será explicado conforme a sua necessidade. O objetivo deste curso não é a linguagem de programação em si, mas sim os métodos numéricos que serão programados nela, portanto alguns elementos da linguagem não serão abordados. Entretanto recomenda-se fortemente que se aprofunde ao máximos nos recursos da linguagem, pois os benefícios serão grandes.

A linguagem *Python* foi idealizada no final da década de oitenta por Guido Van Rossum do *Centrum Wiskunde & Informatica* (Centro de Matemática e Ciência da Computação) em Amsterdam, na Holanda. É a sucessora de uma linguagem chamada ABC. De acordo com o índice TIOBE (*http://en.wikipedia.org/wiki/TIOBE_index*), está entre as dez linguagens mais utilizadas no mundo. Sua primeira implementação data de dezembro de 1989. A versão 2.0 foi lançada em outubro de 2000.

O *Python* é uma linguagem de uso geral cuja filosofia de projeto enfatiza a clareza e legibilidade, assim a sua sintaxe é simples e descongestionada. Além disso possui outras

características:

- **É multiplataforma.** Isto significa que programas escritos nessa linguagem podem ser executados em computadores que rodam *Windows*, *Macintosh*, Maquina Virtual Java, *Linux* e *Android* (celular);
- **O interpretador é *software* livre.** O *interpretador* é o *software* que converte o texto da linguagem *Python* para linguagem de máquina. Ele pode ser instalado em qualquer computador, podendo ser usado até para fins comerciais, sem infringir direitos autorais;
- **É extensível.** Isto significa que a linguagem pode ser acrecida de novas funções e estruturas caso seja necessário;
- **Possui uma comunidade de apoio.** Além do site principal (*http://www.python.org*), possui uma comunidade brasileira, o *Python Brasil* (*http://www.python.org.br/wiki*) onde se podem achar manuais e tutoriais para diversos fins. Possui, também, comunidades nas redes sociais;
- **Possui extensões para programação em várias áreas.** Dentre as várias áreas destacam-se:
  1. **Extensões para Ciência.** Matemáticos, engenheiros e cientistas podem utilizar bibliotecas de Otimização, Equações Diferenciais etc. disponibilizadas em *SciPy*. Além dele existe o *SfePy* (*Simple finite elements in Python*) para o Método dos Elementos Finitos, o *BioPython* para biologia, *AstroPy* para astronomia e o *Rpy* para estatística.
  2. **Extensões para ensino.** O *Python* está incluído no projeto *One Laptop per Child* (um *laptop* por criança, OLPC). Parte do ambiente *Sugar* dessa plataforma foi desenvolvida nessa linguagem. Os programas educacionais *GCompris* podem ser personalizados em *Python.* Há o curso de programação para crianças *Guido van Robot* (GvR), além de outros projetos como o *KineticsKit* (curso de cinemática), *PyGeo* (curso de geometria), e *PyRo* (ensino de robótica).
  3. **Extensões para Internet.** Existe pelo menos uma dúzia de ferramentas para para desenvolvimento de portais e aplicações para internet. Entre essas ferramentas convém mencionar o Zope/Plone, o Turbogears, o Django, e o Pylons.
  4. **Extensões para gestão empresarial.** O *Stoq*, sistema para gestão de empresas comerciais, pode ser personalizado usando *Python*, assim como o *ERP5*.
  5. **Extensões para dispositivos móveis.** O *Python* pode ser usado no *smartphone* Série 60 (S60) da Nokia, no *Playstation Plus*, no *PocketPC* e no *Maemo*.
  6. **Extensões para Multimídia e Entretenimento.** Tem-se o *PyGame* para o desenvolvimento de jogos ou interfaces gráficas que não dependem de nenhum sistema de janelas, e o *PyMedia* para manipulação de arquivos de som e vídeo.

- **Suporta vários paradigmas**. O *Python* é consistente com vários paradigmas de programação principalmente, mas não limitado a, Programação Orientada a Objeto, Programação Funcional, Tipos Dinâmicos, Alocação Dinâmica Automática de Memória, Linguagem de *Script* e Programas *Standalone* (executáveis).

O *Python* 2 e o *Python* 3

Atualmente a versão é um quesito importante que deve ser observado. Existem duas versões da linguagem que são parcialmente **incompatíveis entre si**: a versão 2, chamada muitas vezes de *Python 2*, e a versão 3, chamada de *Python 3*. A versão é expressa por três números separados por um pontos, o primeiro número é a versão principal (2 ou 3), o segundo número é o lançamento, quanto maior o número mais atual é o lançamento, e o terceiro número se refere a pequenas correções que podem ocorrer dentro do lançamento, chamados de *bug fixes*, que não tem explicitamente nenhuma repercussão para o usuário. Frequentemente esse terceiro número é omitido quando se faz referência à versão que está sendo usada.

A versão 2 é a versão antiga que está sendo substituída pela versão 3. Entretanto existe uma base de aplicativos muito grande que ainda não migrou para a versão 3, em particular as bibliotecas `NumPy` e `SciPy` não foram totalmente convertidas para a nova versão 3. Isso faz com que grande parte dos engenheiros e matemáticos industriais, que utilizam essas bibliotecas, continuem a utilizar a versão 2.

Já foram lançadas as versões 2.7.5 e 3.4.0. Para minimizar a defasagem entre as duas versões existe uma biblioteca chamada `__future__` que acrescenta à versão 2 os recursos que estão na versão 3. Este texto aborda a versão 2.6, essa é a que está instalada no Laboratório de Simulação Matemática. Quando necessário serão usados os recursos da biblioteca `__future__` para que a futura migração para a versão 3 seja natural. Ainda para facilitar a migração futura, os recursos da versão 2 que não exitem mais na versão 3 serão omitidas, para que não se corra o risco de tentar usar, na versão 3, alguma coisa que não esteja mais disponível.

O Terminal do *Python*

Para explorar os recursos do *Python* um bom lugar para começar é dentro do Terminal do Interpretador de Comandos. Esse terminal permite que se digite os comandos interativamente e se veja os resultados imediatamente. Permite, ainda que se busque ajuda sobre o uso dos recursos da própria linguagem.

Algumas convenções adotadas nesse texto

Quando se tratar de elementos da linguagem *Python* será usada uma fonte `monoespassada`. Na descrição geral de um comando, as partes que devem ser alteradas pelo programador estarão numa *fonte de largura variável e em itálico*. Por exemplo onde se apresenta a descrição do comando

`print('`*uma string*`')`

qualquer uma das formas abaixo será aceitável

```
print('olá')
print('esta é uma string de teste')
```

pois a palavra `print`, os parêntesis e os apóstrofes são a parte obrigatória do comando, e a expressão *uma string* é a parte configurável.

Toda vez que o símbolo indicador `>>>` aparecer, isso indica que se está dentro de um terminal do *Python*. O que aparecer em seguida estará em negrito e indica que a linha deve ser digitada e em seguida pressionada a tecla *enter* em muitos teclados é representado pelo símbolo "↵". Assim as linhas

```
>>> print('bom dia')
bom dia
```

indicam que o texto `print('bom dia')` deve ser digitado no terminal, pois está em negrito e vem depois do indicador `>>>`, e depois dessa linha ser digitada deve-se pressionar a tecla enter. A linha seguinte `bom dia`, que não está em negrito, mostra o *resultado* da execução do comando da linha anterior.

Entrando e Saindo do Terminal

Para entrar no Terminal do Interpretador do *Python*, se não tiver uma opção no menu, abra um terminal e digite:

```
c:\ python
```

e em seguida pressione a tecla *enter* (↵). Após algum texto sobre a versão e direitos autorais, aparecerá na tela a indicação

```
>>>
```

Isso indica que o interpretador está parado esperando que se digite um comando. Para testar digite:

```
>>> 2+2
4

>>>
```

uma linha com o resultado da fórmula, e outra linha com o indicador de espera para o próximo comando.

Durante a digitação de uma linha é necessário tomar um cuidado especial com os espaços em branco. Os espaços em branco antes do primeiro caractere tem significado especial, são interpretados como recuos na linha e só podem ser usados quando o comando permitir. Por outro lado espaços em branco entre os demais elementos de uma linha normalmente são ignorados, isto é, vários espaços em branco são tratados como apenas um.

Para sair do interpretador, digite

```
>>> quit()
```

É necessário que se digite o abre e fecha parêntesis. Com esse comando o terminal

do interpretador será fechado e o controle do teclado e mouse retorna ao sistema operacional.

Importante. Cada entrada e saída do terminal chama-se de uma *seção* do terminal. Tudo que não é explicitamente gravado em disco ou outro dispositivo é perdido quando uma seção é encerrada, além disso exitem certas configurações (como importação de módulos) que só precisam ser feitas uma única vez a cada seção. É importante ficar atento a esse detalhe ao longo do uso do terminal.

Explorando os Conceitos Básicos de Programação Usando o Terminal

A seguir são descritos os elementos básicos necessários para programação que são basicamente: Os tipos de dados, operações aritméticas, variáveis, funções e importação de bibliotecas.

Tipos Básicos de Dados

Existem quatro tipos de dados básicos que são classificados da seguinte forma: dados numéricos, strings, dados lógicos e o conjunto vazio.

Dados Numéricos

Dados numéricos ou números podem ser escritos como uma série de dígitos, com ou sem um *ponto decimal*, ou na *notação E* (A notação E será descrita em ***). Na língua portuguesa o separador das casas decimais é a vírgula, entretanto a linguagem de programação segue a representação da língua inglesa que utiliza o ponto como separador para as casas decimais.

Os números são normalmente escritos no sistema decimal, cabendo ao próprio computador a conversão de decimal para binário, e vice e versa. Os dados numéricos são de quatro tipos: Inteiros Simples, Inteiros Longos, Números Reais (comumente chamados de números de *ponto flutuante*), e Números Complexos (com uma parte real e outra imaginária).

Inteiros Simples

Números escritos sem o ponto decimal. Ele pode ser positivo ou negativo. O sinal correspondente é opcional para os números positivos: -8 0 2 10 etc.

Internamente o computador reserva uma quantidade fixa de memória para um inteiro simples, normalmente oito *bytes*, o que faz com que um inteiro simples esteja limitado apenas a números entre -2.147.483.647 e +2.147.483.647.

Inteiros Longos

Não possuem casas decimais, e são muito grandes (maiores, em módulo, que 2.147.483.647). Um inteiro longo é representado pela letra L (tanto maiúsculo como minúsculo, entretanto normalmente se usa maiúsculo para não confundir com o número 1) no final: 200L 3L 1234578L

Existe uma diferença substancial entre os inteiros simples e os longos. Para os

inteiros longos, o computador usa a quantidade de *bytes* necessária para guardar o número. Assim inteiros longos não tem limitação de tamanho (apenas a limitação da memória do computador), em compensação operações aritméticas com inteiros longos são consideravelmente mais lentas.

Números Reais

Números com casas decimais: 2.0 0.3e11 0.3E11 9.0e-4 etc. Note que a casa decimal é marcada por um ponto, e não por uma vírgula.

A distinção entre números inteiros e números reais não é uma mera formalidade, o computador trata internamente cada um desses números de forma *muito* diferente. O espaço de memória ocupado por um número inteiro simples é menor do que o espaço ocupado por um número real, além disso as operações aritméticas com números inteiros é muito mais rápida do que uma operação aritmética com números reais.

Números Complexos

Números com uma parte real e uma imaginária. A parte imaginária é marcada colocando-se a letra j ou J (que representa $\sqrt{-1}$): 1j 2-4j 1.45+3.3j.

Strings

São encadeamentos alfanuméricos, isto é sequências de letras e números formando um texto. Para indicar que os dados são do tipo *string*, eles são colocados entre apóstrofos ou aspas: Exemplo: `'x'`, `'Intensidade da força'`, “`entre com a velocidade`”, etc.;

Dados Lógicos

Representam condições que podem ser falsas ou verdadeiras, a sua representação é `False` (para falso) e `True` (para verdadeiro);

O Conjunto vazio

O equivalente computacional ao Conjunto Vazio chama-se `None`.

Ao digitar as palavras `True`, `False` e `None` é importante que a primeira letra seja maiúscula. Essas palavras são diferentes de `true`, `false` e `none` (sem as iniciais maiúsculas) que não tem nenhum significado especial.

Exercícios.

1. Ache os erros nos seguintes números:

+12	+123.e01	99E2.5	126	.0006
123,245	R$125,00	$12.00	3j	3.0L

Comandos e Erros de Sintaxe

No terminal digita-se um comando e pressiona-se a tecla enter. Entretanto para se dar um comando é necessário que esse comando siga uma estrutura pré determinada para que o computador possa compreender o comando. A essa estrutura chama-se de *sintaxe*. Caso alguma coisa não seja compreendida pelo interpretador, este emitirá uma mensagem de erro. A forma geral de uma mensagem de erro é:

```
Traceback (most recent call last):
  File "nome do arquivo", número da linha, in nome do módulo
Nome do erro: detalhes do erro.
```

A última linha da mensagem fornece o nome e os detalhes do erro. A seguir são apresentados dois exemplos de erros: No primeiro digita-se o comando `none`, como ele não existe gera-se um `NameError` e aparece a mensagem "`name 'none' is not defined`"; No segundo tenta-se fazer a divisão do número 1 por zero, que gera um `ZeroDivisionError`, isto é, um erro de divisão por zero.

```
>>> none
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'none' is not defined
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

As mensagens de erro são fontes importantes para ajudar a descobrir problemas nos programas durante a fase de desenvolvimento.

Digitação de linhas longas

O símbolo "\" (barra invertida) tem a função de indicar ao terminal que um comando digitado não cabe na linha, e que a linha seguinte é uma continuação da linha que termina com a barra.

Operações Aritméticas

As operações aritméticas são essenciais na programação de computadores, pois é a partir delas que cálculos são feitos, condições são testadas e decisões são tomadas. Existem oito operações aritméticas que podem ser usadas: adição, subtração, multiplicação, divisão, divisão truncada, resto da divisão, potenciação e negativação. A sua representação é semelhante à representação que se faz uso na matemática. Mais detalhadamente tem-se:

Adição

Usa-se o símbolo + para representar a adição, por exemplo:

```
>>> 2+2
4
>>> 3.8 + 1.2
5.0
>>> 2+3j+1-5j
(3-2j)
```

Se as duas parcelas da soma forem números inteiros (sem casas decimais), então o resultado será um número inteiro. Se **ao menos uma** das parcelas for um número real (com casas decimais), então o resultado será um número real, mesmo se as casas decimais forem zero.

As últimas duas linhas mostram que a soma pode ser feita normalmente com números complexos.

Subtração

Usa-se o símbolo − para representar a subtração, por exemplo:

```
>>> 2-1
1
>>> 3.8 - 1.2
2.5999999999999996
>>> 2j-(1+1j)
(-1+1j)
```

Note que nos exemplos, a representação do resultado da operação com números inteiros é sempre exata, enquanto que no segundo exemplo a solução que deveria ser 2.6 não foi exata. Essa é um problema comum com números reais que se deve ter sempre em mente.

Multiplicação

Usa-se o símbolo * para representar a multiplicação, por exemplo:

```
>>> 2*3
6
>>> 3.8 * 1.2
4.559999999999999
>>> (2+1j)*1j
(-1+2j)
```

Divisão

Usa-se o símbolo / para representar a divisão, onde o primeiro fator representa o numerador (o número que será dividido) e o segundo fator o denominador (ou divisor).

Se os dois fatores da divisão forem números inteiros, a operação retornará apenas

a parte inteira da divisão, desprezando a parte fracionária.

```
>>> 56/3
18
```

Que deveria resultar em 18.6666.... Como isso não é um comportamento que normalmente se espera de operação de divisão, isso foi corrigido no *Python* 3. Para se corrigir isso deve-se, invocar a biblioteca `__future__` com o comando

```
>>>from __future__ import division
```

essa linha tem que ser digitada apenas uma vez a casa seção.

Assim:

```
>>> 5/2
2
>>> 56/3
18
>>> from __future__ import division
>>> 5/2
2.5
>>> 56/3
18.666666666666668
```

Caso não se queira usar a biblioteca `__future__` pode-se usar o artifício de fazer com que ao menos um dos fatores tenha uma casa decimal, assim o resultado será o número completo, por exemplo:

```
>>> 56.0/3
18.666666666666668
>>> 56/3.0
18.666666666666668
>>> 56.0/3j
-18.666666666666668j
```

Cuidado! - Sempre que houver uma operação de divisão, deve-se usar o módulo `division` da biblioteca `__future__` antes da primeira vez que se usar o operador de divisão. Caso isso não seja feito corre-se o risco das contas não darem o resultado que se espera. A forma de fazer isso é digitando a linha:

```
>>> from __future__ import division
```

Divisão Truncada

Usa-se o símbolo `//` (duas barras) para representar a divisão que retorna apenas a parte inteira do quociente, mesmo que os números dados sejam números reais. Como na divisão comum, o primeiro fator representa o numerador (o número que será dividido) e o segundo fator o denominador (ou divisor). Se ao menos um dos números for um número real, então o resultado será um número real com as casas decimais zero. Por exemplo:

```
>>> 56//3
18
>>> 56.0//3
18.0
>>> 56//3.0
18.0
```

Resto da Divisão

Usa-se o símbolo `%` (porcentagem) para representar o resto da divisão de dois números inteiros. Como na divisão comum, o primeiro fator representa o numerador (o número que será dividido) e o segundo fator o denominador (ou divisor). Se ao menos um dos números for um número real, então o resultado será um número real com as casas decimais zero. Por exemplo:

```
>>> 56%3
2
>>> 56.0%3
2.0
>>> 56%3.0
2.0
```

Potenciação

Usa-se o símbolo `**` para representar a potenciação, onde o primeiro número é a base e o segundo é o expoente, por exemplo:

```
>>> 2**3
8
>>> 3**2
9
```

Negativação ou Oposto

A negativação é a transformação de um número positivo em um negativo ou vice e versa. Usa-se o símbolo − para representar a negativação. Apesar de usar o mesmo símbolo da subtração, a negativação difere da subtração no que se refere ao número de parcelas, que

no caso da negativação possui apenas uma. Por exemplo

```
>>> -2
-2
>>> -2.3
-2.3
```

Expressões, Prioridades e Parêntesis

Uma expressão é uma combinação de operadores aritméticos, operandos e parêntesis, arranjados numa forma significativa, de tal modo que, quando calculados resulta um valor numérico (existem, entretanto, exceções em que o valor retornado pode não ser numérico).

Ao se ter várias operações em uma única expressão, as operações não são feitas da esquerda para a direita, elas seguem uma ordem determinada pelas prioridades das operações. Caso se deseje alterar a ordem natural das prioridades usam-se os parêntesis para indicar quais operações deverão ser feitas primeiro. A ordem de prioridade das operações é:

1º – Parêntesis;

2º – Negativação;

3º – Potenciação;

4º – Multiplicação e Divisão;

5º – Soma e Subtração.

Dentro de um mesmo nível, as operações são feitas da esquerda para a direita.

Exemplos:

```
>>> 2+3*4
14
```

Note que a multiplicação foi feita primeiro (resultando em 12) pois tem uma prioridade maior, em seguida foi feita a soma.

```
>>> (2+3)*4
20
```

Nesse caso a soma foi feita primeiro (por causa dos parêntesis) e a multiplicação foi feita depois.

```
>>> 3+-2
1
```

Apesar da sequência de sinais a expressão faz sentido. Somou-se -2 ao número 3.

Modos das expressões (caso não se use o módulo `__future__` no *Python* 2)

Uma expressão pode ser escrita em dois modos: no modo inteiro ou no modo misto. Ela estará escrita no modo inteiro se todas as parcelas forem inteiras. No modo inteiro todos os cálculos intermediários serão inteiros, desprezando as casas decimais caso elas

ocorram.

Ela estará escrita no modo misto se ao menos uma das parcelas for um número real. Os cálculos serão feitos no modo inteiro até chegar à parcela com número real. A partir desse ponto os cálculos serão feitos com números reais.

Como exemplo, para ter o resultado de

$$\frac{(3+4)^2}{6+7}$$

Escreve-se:

```
>>> (3.0+4)**2/(6+7)
3.769230769230769
```

Note que os parêntesis no denominador são obrigatórios, caso contrário a divisão do numerador por seis seria feita primeiro, e ao resultado seria somado sete, que daria um resultado diferente. Foi adicionada a casa decimal com zero no número 3 (poderia ter sido em qualquer um dos números, ou em todos) para forçar a conversão do resultado para um número real e as operações subsequentes ocorrerem no modo misto. Caso isso não fosse feito o resultado seria 3 (a parte inteira do resultado).

Exercícios.

2. Digite no terminal as seguintes equações:

$\frac{1}{2}\frac{2}{3}\frac{3}{4}\frac{4}{5}$ $\qquad$ $\frac{3^2-2}{3-2^2}$ $\qquad$ $\frac{1}{2}+\frac{2}{3}+\frac{3}{4}+\frac{4}{5}$

$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\frac{1}{2}}}}$ $\qquad$ 2^{2^3} $\qquad$ $2^{2\times 3}$

Variáveis e o comando de atribuição (ou definição)

Muitas vezes é necessário guardar uma informação na memória para ser usada em um cálculo subsequente, gravar num arquivo ou até enviar pela internet para outro local. Esse é o objetivo da variável. Uma *Variável* é um espaço na memória reservada para guardar uma unidade de informação. Possui esse nome porque a informação guardada pode ser alterada, isto é, pode variar.

Toda variável tem:

- *Um nome* - que é como ela será chamada ao guardar e recuperar a informação da memória;
- *Um tipo* - esse tipo (inteiro, real etc.) é definido quando variável é criada, baseando-se no conteúdo que foi a ela atribuído; e
- *Uma abrangência (ou escopo)* - uma variável só pode ser usada (para

armazenar e resgatar dados) no módulo do programa que foi criada.

Para criar uma variável usa-se o comando de atribuição

nome da variável = *expressão*

Uma variável só pode ser usada depois de criada através de um comando de atribuição. A tentativa de usar uma variável que ainda não foi criada gerará uma mensagem de erro no terminal.

Uma vez que a variável é criada, ela pode fazer parte de uma expressão como um número qualquer. O modo da expressão dependerá do tipo de dado que foi utilizado no último comando de atribuição.

Exemplo:

```
>>> x = 2
```

Guarda o número 2 (número inteiro) numa variável chamada `x`. A partir desse ponto toda vez que se usar `x` numa expressão, o `x` será computado como sendo 2. Por exemplo:

```
>>> x+3
5


>>> y = 2.1
>>> y+8.9
11.0
```

No exemplo acima foi criada uma variável `y` que contém o número real 2.1. A soma de y com 8.9 dá o número 11 como número real.

```
>>> z+3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
```

Nesse caso como a variável `z` não foi criada, não é possível realizar a conta, assim o terminal gera uma mensagem de erro avisando que a variável `z` não está definida.

Uma variável criada em uma seção do terminal será perdida quando a seção se encerrar. Assim caso se inicie uma nova seção, essa não reconhecerá as variáveis utilizadas em seções anteriore.

Atribuição Aumentada

Uma vez que uma variável é criada por um comando de atribuição, o seu conteúdo pode ser modificado, não só atribuindo novos valores diretamente por uma nova atribuição, mas acrescentando, retirando valores etc., usando comandos de atribuição aumentados. O comando de atribuição aumentado mais comum é o incremento, simbolizado pelo símbolo `+=` . O comando

```
x += y
```

é o mesmo que `x = x + y`, isto é,diz ao computador: Some ao que já está na variável *x* o valor *y*. O exemplo a seguir cria uma variável chamada contador, e depois incrementa duas vezes, tendo como resultado final o número 2.

```
>>> contador = 0
>>> contador += 1
>>> contador += 1
>>> contador
2
```

Além do incremento existem outras variações da atribuição aumentada que estão listadas na tabela abaixo:

Atribuição	Equivalente
x += y	*x = x+y*
x -= y	*x = x-y*
*x *= y*	*x = x*y*
x /= y	*x = x/y*
x %= y	*x = x%y*

A variável "_".

Dentro do terminal, existe uma variável, cujo nome é o caractere sublinhado (_) que significa: *o resultado da última operação*. Isso facilita a digitação quando várias operações são feitas em sucessão, por exemplo:

```
>>> x=3
>>> x+4
7
>>> _ * 2
14
```

Nomes de Variáveis: Identificadores

O nome de uma variável é um caso particular de Identificador. Identificadores são palavras criadas pelo programador para dar nome a alguma coisa. Essa coisa pode ser uma região na memória, isto é, uma variável, ou o nome de uma função que o programador criou. Para criar um identificador é necessário observar algumas regras:

1. Tem que começar por uma letra ou o caractere sublinhado (_);
2. Só pode ter letras não acentuadas, números e o caractere sublinhado;
3. Maiúsculas e minúsculas são letras diferentes;

4. Não pode ser uma palavra reservada; e
5. Não deve conflitar com nomes de variáveis e funções predefinidas.

Identificadores iniciados por um ou dois caracteres sublinhados tem significado especial, indicando se tratar de ser privativo ao módulo, ou algum objeto do sistema, e devem ser evitados quando não utilizados com essas finalidades.

Exemplos:

Os identificadores abaixo são permitidos, além disso As variáveis `a2` e `A2` são variáveis diferentes.

```
x    a_b_c    a2    A2
```

Os identificadores abaixo não são permitidos.

`2a` - Começa com número

`goiás` - Possui uma letra acentuada

`chave vertical` - Possui espaço em branco. (são de fato duas palavras, cada uma individualmente poderia ser um identificador)

`if` - É palavra reservada da linguagem.

São palavras reservadas, que não podem ser usadas como identificadores:

```
as          if          in

is          or          and

def         del         for

not         try         elif

else        exec        from

pass        with        break

class       print*      while

assert      except      import

lambda      global      raise

return      yield       finally
```

`continue`	* No *Python* 3 o a palavra `print` deixa de ser uma palavra reservada.

São identificadores a serem evitados porque são comumente usadas para outros fins:

`id`	`all`	`any`	`bin`	`chr`
`cmp`	`dir`	`hex`	`int`	`len`
`oct`	`ord`	`pow`	`set`	`str`
`sum`	`zip`	`bool`	`dict`	`eval`
`exit`	`file`	`hash`	`help`	`iter`
`list`	`long`	`math`	`next`	`open`
`quit`	`type`	`vars`	`apply`	`bytes`
`cmath`	`float`	`input`	`print`	`range`
`round`	`slice`	`super`	`tuple`	`buffer`
`coerce`	`divmod`	`filter`	`format`	`intern`
`locals`	`object`	`reload`	`sorted`	`compile`
`complex`	`credits`	`getattr`	`globals`	`hasattr`
`unicode`	`callable`	`raw_input`	`complex`	

Funções Matemáticas Predefinidas.

Além das operações aritméticas, existem várias funções que são úteis e são providas pelo ambiente de programação para uso imediato. Essas funções são chamadas de *Funções Predefinidas*.

Dentre as funções predefinidas, algumas são de interesse específico da matemática. Essas funções estão listadas na tabela abaixo juntamente com uma descrição.

`abs(x)`	Retorna o valor absoluto de *x*. O resultado é sempre do mesmo tipo (inteiro ou real) do número *x*.
`int(x)`	Converte um número real *x* em número inteiro descarta a as casas decimais.
`long(x)`	Converte um número *x* em um número inteiro longo. Se *x* for um número real, descarta as casas decimais.

`float(x)`	Converte um número inteiro *x* em real. Preenche as casas decimais com zeros. A palavra "float" vem de uma terminologia muito utilizada em inglês que chama os números reais de números "com ponto flutuante", onde o ponto se refere ao separador de casas decimais.
`complex(x,y)`	Converte dois números em um número complexo onde *x* é a parte real e *y* a parte imaginária.
`round(x)` ou `round(x,n)`	Arredonda um número para o inteiro mais próximo, ou para o número de casas decimais especificado por `n`.
`divmod(x,y)`	Calcula o quociente e o resto da divisão de dois números inteiros *x*/*y*. Retorna o par `(`*quociente*`,` *resto*`)`
`min(`x_1`,` x_2`, …,` x_n`)`	Retorna o menor dos valores dados
`max(`x_1`,` x_2`, …,` x_n`)`	Retorna o maior dos valores dados
`sum((`x_1`,` x_2`, …,` x_n`))` ou `sum([`x_1`,` x_2`, …,` x_n`])`	Retorna a soma dos valores dados. Note que no caso da função `sum` são necessário os dois parêntesis ou parêntesis com colchetes. O motivo disto será visto mais tarde.

Exemplos:

```
>>>abs(2)
2
>>>abs(-2)
2
>>>abs(-2.0)
2.0
```

A função `abs` mantém o tipo do número, se o número dado for inteiro o resultado é um número inteiro, se o número dado for real, o resultado será um número real.

```
>>> abs(1-1j)
1.4142135623730951
```

Para um número complexo o resultado é o módulo do número complexo no plano Real x Imaginário.

Mudança de tipos de números

As funções `int(...)`, `long(...)`, `float(...)` e `complex(...)` são utilizadas para a conversão dos tipos dos dados. Para se deseja usar um número como real basta colocar um ponto para indicar a casa decimal. Entretanto esse procedimento não pode ser usado se na expressão aparece uma variável. Para usar uma variável sabidamente inteira como se fosse real pode-se usar a função `float(...)`. Seguem a seguir alguns exemplos de conversão de tipos.

```
>>> float(2)
2.0
>>> x = 5
>>> float(x)/2
2.5


>>> int(2.1)
2
>>> int(2.9)
2
```

O resultado é um número inteiro.

Arredondamentos

A função `round(...)` arredonda um número.

```
>>> round(2.7)
3.0
>>> round(2.4)
2.0
```

O resultado da função `round(x)` é um número real. Caso seja necessário que o resultado final seja um número inteiro, essa função deve ser composta com a função `int(x)`.

```
>>> int(round(2.4) )
2
```

Um segundo parâmetro especifica o número de casas decimais a ser arredondada.

```
>>> round(1.0/3,2)
0.33000000000000002
```

Funções matemáticas diversas

```
>>> divmod(7,3)
(2, 1)
>>> divmod(8,3)
(2, 2)
```

O primeiro número é a parte inteira da divisão e o segundo número o resto da

divisão.

```
>>> min(5,2,8,7)
2
>>> max(5,2,8,7)
8
```

O menor valor é 2 e o maior valor é 8.

```
>>> x = 5
>>> y = 10
>>> z = 8.3
>>> max(x,y,z)
10
```

Podem ser usadas variáveis nas operações aritméticas e funções.

```
>>> sum((5,2,8,7))
22
```

Note os parêntesis duplos, se usar parêntesis simples dará uma mensagem de erro.

Importação de Módulos e Funções Matemáticas.

Para poder usar funções matemáticas como seno, cosseno, raiz quadrada, etc. é necessário importá-las de uma biblioteca de funções.

Bibliotecas de funções, ou mais comumente chamadas *módulos*, são arquivos que guardam um conjunto de funções e variáveis com um fim específico. Existem módulos para funções matemáticas, para gerenciar bancos de dados, para comunicação com a internet etc.

A importação de um módulo só precisa ser feita **uma única vez** durante uma seção. Comumente a importação é feita no inicio da seção, entretanto ele pode ser feita em qualquer momento.

As funções matemáticas encontram-se na biblioteca, ou módulo, chamado `math`. Existem duas maneiras de se importar uma função a partir de um módulo, uma importação direta, e outra indireta que são descritas a seguir.

Importação Direta usando `from..import`.

Na importação direta apenas a função, ou funções, que se deseja usar são importadas. O comando geral para a importação direta é:

`from` *módulo* `import` *função*$_1$`,` *função*$_2$`,... ,` *função*$_n$

Onde "*módulo*" é o nome do módulo que contém as funções que se deseja importar e "*função*$_1$" , "*função*$_2$", etc. são os nomes das funções.

No exemplo a seguir importa-se diretamente a função `sin` (seno) e a variável `pi` (pi=3,1416...) do módulo `math`. Primeiramente tenta-se usar a função seno e pi sem importar e depois calcula-se seno(π/4).

```
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
>>> sin(pi/4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>> from math import sin, pi
>>> pi
3.1415926535897931
>>> sin(pi/4)
0.70710678118654746
```

É possível importar todas as funções e variáveis de uma biblioteca usando o comando[1]

`from` *módulo* `import *`

Onde a lista de funções é substituída pelo asterisco que indica "todos". No exemplo a seguir todas as funções do módulo `math` são importados e as funções seno e cosseno são usadas juntamente com a variável `pi`.

```
>>> from math import *
>>> cos(pi/4)
0.70710678118654757
>>> sin(pi/4)
0.70710678118654746
```

Cuidado. Use essa forma de importação apenas quando estiver no terminal e mesmo assim quando o módulo a ser importado é feito para ser usado dessa forma, pois pode acontecer de se importar uma função ou variável com o mesmo nome de uma já criada. Chamamos essa situação (duas funções ou variáveis diferentes com o mesmo nome) de "conflito de nome" e fica difícil saber qual das duas funções é a que está disponível.

Importação Indireta usando `import`.

Na importação **indireta** todo o módulo é importado e as funções e variáveis são acessadas a partir do módulo usando a notação *módulo*`.`*função* (módulo ponto função). Assim a função `sin(x)` do módulo `math` é chamada de `math.sin(x)`.

O comando usado para a importação indireta é:

1 Uma exceção a essa regra é o módulo `__future__`, uma tentativa de importar todo o módulo usando * irá resultar em uma mensagem de erro.

`import` *módulo*

O exemplo a seguir ilustra a importação indireta calculando onde se calcula $\text{seno}\left(\frac{\pi}{4}\right)$.

```
>>> import math
>>> math.sin(math.pi/4)
0.70710678118654746
```

Essa é a maneira mais segura de se importar um módulo pois evita os conflitos de nomes. Normalmente o problema com esse tipo de importação é que os nomes das funções e variáveis podem ficar muito extensas.

Existe uma variação do comando de importação que ao mesmo tempo importa e muda o nome do módulo. Normalmente isso é usado para abreviar os nomes dos módulos com o objetivo de diminuir a quantidade de caracteres digitados. A variação do comando é:

`import` *módulo* `as` *novo nome*

O exemplo anterior usando o recurso de renomear fica:

```
>>> import math as m
>>> m.sin(m.pi/4)
0.70710678118654746
```

Dessa forma pode-se abreviar um pouco os nomes das funções e variáveis importadas.

As funções matemáticas do módulo `math`.

Uma vez importado o módulo `math`, nas fórmulas e cálculos podem ser usadas as seguintes funções matemáticas:

Funções Diversas	
`sqrt(x)`	Calcula $\sqrt{x}$
`pow(x,y)`	Potenciação: x^y (equivale a *x****y*)
`hypot(x,y)`	Calcula $\sqrt{x^2+y^2}$
`factorial(x)`	Calcula o fatorial de *x*. (O resultado é um número inteiro)
`copysign(x,y)`	Retorna *x* com o sinal de *y*.

`fabs(x)`	Calcula o módulo de *x*. Difere da função predefinida `abs(x)` no sentido de que sempre retorna um número real.
`fmod(x,y)`	Calcula o resto da divisão de *x* por *y*. Apesar do resto da divisão ser um número inteiro, essa função retorna um número real, para se obter um inteiro é necessário fazer a composição `int(fmod(x,y))`.
`modf(x)`	Decompõe o número *x* em sua parte fracionária, e sua parte inteira. O resultado de cada um é colocado na forma de número real, com o sinal do número original.
`fsum((`x_1`, `x_2`, …, `x_n`))`	Soma os números x_1, x_2 até x_n. O resultado é sempre um número real.

Trigonométricas	
`pi`	A constante irracional 3.1415926535897931 ...
`cos(x)`	Cosseno de *x*
`sin(x)`	Seno de *x*
`tan(x)`	Tangente de *x*
`acos(x)`	Ângulo cujo cosseno é *x*
`asin(x)`	Ângulo cujo seno é *x*
`atan(x)`	Ângulo cuja tangente é *x*
`atan2(x,y)`	Ângulo cuja tangente é *x*/*y*
`degrees(x)`	Converte um ângulo *x* de radianos para graus
`radians(x)`	Converte um ângulo *x* de graus para radianos

Arredondamentos	
`trunc(x)`	Arredonda o número *x* em direção ao zero (Toma apenas a parte inteira do número) O resultado é um número real, para convertê-lo a um número inteiro é necessário fazer a composição `int(trunc(x))`.
`ceil(x)`	Arredonda o número *x* em direção a + infinito. O resultado é um

	número real, para convertê-lo a um número inteiro é necessário fazer a composição `int(ceil(x))`.
`floor(x)`	Arredonda o número *x* em direção a -infinito. O resultado é um número real, para convertê-lo a um número inteiro é necessário fazer a composição `int(floor(x))`.

Exponenciais e Logaritmos	
`e`	A constante irracional 2.7182818284590451 ...
`exp(x)`	e^x
`log(x)`	Logaritmo de *x* na base *e*
`log10(x)`	Logaritmo de *x* na base 10
`log(x,b)`	Logaritmo de *x* na base *b*
`log1p(x)`	$\log(1+x)$
`ldexp(m,p)`	Calcula o número dado pela mantissa, *m*, e expoente, *p*, pela fórmula $m \cdot 2^p$
`frexp(x)`	Decompõe o número *x* em duas partes: mantissa, *m*, e expoente, *p*, tal que $x = m \cdot 2^p$

Hiperbólicas	
`cosh(x)`	Cosseno hiperbólico
`sinh(x)`	Seno hiperbólico
`tanh(x)`	Tangente hiperbólica
`acosh(x)`	Arco-cosseno hiperbólico
`asinh(x)`	Arco-seno hiperbólico
`atanh(x)`	Arco-tangente hiperbólica

Outros Módulos Matemáticos

O módulo `math` é o módulo padrão da linguagem, entretanto ele possui a limitação de que as suas funções não trabalham com números complexos. Para se trabalhar

com números complexos pode-se usar a biblioteca `cmath`, que também é um módulo padrão da linguagem.

```
>>> from math import sin
>>> sin(1)
0.8414709848078965

>>> sin(1j)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float

>>> from cmath import sin
>>> sin(1j)
1.1752011936438014j
```

Entretanto a melhor opção para se trabalhar com funções matemáticas é o módulo `numpy`[2] (Abreviação de *Numerical Python*). Esse é um módulo que tem que ser instalado separadamente da linguagem, entretanto ele possui algumas vantagens: Os cálculos internos das funções são otimizadas, dando resultados mais rápidos, aceita números complexos, e aceita ainda listas de valores como parâmetros. Caso o `numpy` esteja instalado no sistema computador, essa é a melhor opção quando se trata de funções matemáticas. O exemplo a seguir ilustra o uso do `numpy`.

```
>>> from numpy import sin, pi
>>> sin(pi/4)
0.70710678118654746

>>> sin(1j)
1.1752011936438014j

>>> sin((1,2,3))
Array([ 0.84147098,  0.90929743,  0.14112001])
```

A palavra `Array` que aparece no resultado será abordado mais tarde quando se tratar de dados estruturados.

A Função Declaração

Além das funções predefinidas e das funções importadas, o programador pode criar as suas próprias funções. Existem dois tipos de funções diferentes: a *Função Declaração*, ou *Função Lambda*, que é formada por uma única fórmula e que retorna um único valor, e a *Função Definição*, que é mais complexa e que pode envolver vários comandos e ter vários efeitos colaterais. A Função Definição será estudada na página 71.

O comando para criar uma Função Declaração é:

nome da função = `lambda` *variável1*`,` *variável2*`,` … `:` *fórmula*

As variáveis que aparecem na definição da função são variáveis *sem efeito*, isto é,

2 Infelizmente até a escrita deste texto o `numpy` não está disponível em *Python* 3 para todas as plataformas, essa que é uma das razões que o *Python* 2 ainda exista.

elas aparecem na expressão apenas para marcar a posição da variável e a sua função na fórmula, não possuindo, portanto nenhum valor particular. O exemplo a seguir cria uma função chamada `cubo` que eleva o número dado ao cubo, em seguida calcula o cubo de 2 e de 3:

```
>>> cubo = lambda x: x*x*x
>>> cubo(2)
8
>>> cubo(3)
27
>>> cubo(3.0)
27.0
```

Note que da maneira que a função foi criada, o tipo do resultado (inteiro ou real) dependerá do tipo da variável dada para a função. A variável `x` é uma variável sem efeito, o seu valor só vai ser determinado na hora em que a função é chamada. O exemplo a seguir cria a função `area` que calcula a área de um círculo a partir do raio:

```
>>> from math import pi
>>> area = lambda r: pi*r*r
>>> area(1)
3.141592653
>>> area(2)
12.566370612
>>> area(3.0)
28.274333877
```

O exemplo a seguir cria a função chamada `montante` que calcula a fórmula $p\left(1+\frac{i}{100}\right)^n$, em seguida calcula um valor para p=100, i=10 e n=2. Note que essa função possui três parâmetros: *p*, *i* e *n*.

```
>>> montante = lambda p,i,n: p*(1+i/100.0)**n
>>> montante(100,10,2)
121.00000000000001
```

Objetos, propriedades e métodos

Numa definição simples, objetos são variáveis que são compostas por partes. Essas partes podem ser de dois tipos: propriedades ou métodos. Chama-se de *propriedade* quando essa parte tem a função de armazenar algum valor, como se fosse uma variável, e de *método* quando essa parte é executável, como se fosse uma função, podendo retornar, ou não um valor.

A forma de se fazer referência a uma parte de um objeto é através da notação de

ponto:

nome do objeto **.** *nome da propriedade*

ou

nome do objeto **.** *nome do método* (*argumentos*)

Um número complexo é um exemplo bem simples de objeto. Um número complexo possui duas propriedades: a propriedade `real`, que é a parte real do número, e a propriedade `imag`, que é a parte imaginária. Além disso possui um método, o método `conjugate` que retorna o conjugado complexo do número. O exemplo a seguir ilustra o uso das propriedades e métodos de um número complexo.

```
>>> x = 2-1j
>>> x.real
2.0

>>> x.imag
-1.0

>>> x.conjugate()
(2+1j)
```

Outro exemplo de objeto são os módulos, quando são importados de forma indireta. Quando a biblioteca `math` é importada pelo comando

```
import math
```

ela é um objeto. Sendo `math.pi` uma propriedade desse objeto. As funções matemáticas são métodos, assim

```
math.sin(...)
```

é um método do objeto `math`.

Apesar dos nomes diferentes, propriedades e métodos são tratados como variáveis e funções respectivamente. Uma diferença digna de nota é que um método pode alterar as propriedades do objeto ao qual ele pertence, e isso, em geral, é desejável, conforme será visto nos objetos que serão apresentados.

Strings

Frequentemente num programa é necessário trabalhar com textos, como, por exemplo, para solicitar alguma informação do usuário ou para informar o resultado da execução do programa. O tipo de dado usado para armazenar texto chama-se *string*. A palavra vem do inglês que significa, nesse contexto, *encadeamento*. Refere-se ao encadeamento de letras, números, espaços em branco, caracteres especiais e marcas de tabulação e fim de linha.

Para indicar que um texto se trata de uma *string*, ele é colocado entre apóstrofos ou entre aspas, além disso, a exemplo dos números, uma *string* pode ser armazenada em uma variável. Para ilustrar seguem alguns exemplos:

```
>>> uma string completa
```

```
  File "<stdin>", line 1
    uma string completa
             ^
SyntaxError: invalid syntax
```

Ao digitar o texto sem apóstrofes o computador interpreta a linha como se fosse uma fórmula gera uma mensagem de erro indicando sintaxe inválida.

```
>>> 'uma string completa'
'uma string completa'
```

Colocando o texto entre apóstrofes a linha é repetida.

```
>>> x = 'uma string completa'
>>> x
'uma string completa'
```

A *string* pode ser armazenada na variável *x*. Ao se digitar o nome da variável, a *string* que ela contém aparece.

```
>>> '2+2'
'2+2'
>>> 2+2
4
```

Entre apóstrofes a expressão 2+2 é uma *string*. Sem apóstrofes é uma expressão a ser calculada.

Apesar de não indicarem quantidade, duas strings podem ser somadas. O resultado da soma de duas strings é o encadeamento da primeira com a segunda. No exemplo a seguir a variável *z* é a soma das variáveis *x* e *y* que são strings. O resultado é uma string composta pela junção das duas.

```
>>> x = 'Primeira frase'
>>> y = 'Segunda frase'
>>> z = x+y
>>> z
'Primeira fraseSegunda frase'
```

Existem situações em que é necessário converter um número em um texto cujos caracteres são o próprio número. Isso é feito usando a função predefinida str(). O exemplo a seguir ilustra o uso dessa função:

```
>>> x = 3.2
>>> c = 'O número é '
>>> print(c+x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'float' objects
```

Ocorre um erro pois não é possível somar texto com número.

```
>>> print(c+str(x))
O número é 3.2
```

A conversão de número para texto resolve esse problema.

Tipos de dados Estruturados.

Dados de tipo estruturado são objetos, isto é, são dados compostos por partes. Existem três tipos de dados estruturados que são nativos: Listas, Tuplas e Dicionários. E um que é definido dentro do módulo `numpy`: o vetor generalizado (chamado de `array`). Segue uma descrição de cada um.

Listas, Tuplas e Arrays

Tanto as listas, as tuplas como as *arrays* são simplesmente sequências que são tratadas como uma unidade. Elas vão diferir uma da outra em três aspectos: 1) a quantidade de recursos (memória) que consomem, 2) o que se pode fazer com o tipo estruturado uma vez criado, e 3) os tipos de dados que podem ser armazenados.

Listas

A sintaxe para se definir uma Lista é:

nome da lista = [*elemento 1*, *elemento 2*, elemento 3, …]

O exemplo a seguir cria uma lista, chamada `x`, de números de 0 a 0,5 com incrementos de 0,1:

```
>>> x = [0, 0.1, 0.2, 0.3, 0.4, 0.5]
```

O exemplo a seguir cria uma lista, chamada `semanas`, com os nomes dos dias da semana. A barra invertida ao final da linha informa ao terminal que o comando continua na próxima linha:

```
>>> semanas = ['Segunda-feira', 'Terça-feira', \
'Quarta-feira',  'Quinta-feira', 'Sexta-feira', \
'Sábado', 'Domingo']
```

Os elementos de uma lista não precisam ser do mesmo tipo, o exemplo a seguir cria uma lista, chamada `funcionario`, formada por elementos com tipos diferentes: O primeiro elemento é o nome (tipo string), o segundo a idade (tipo inteiro) e o terceiro elemento é o salário (tipo real).

```
>>> funcionario = ['João', 35, 1320.18]
```

Aumentando uma lista existente como método `append(...)`

Uma vez criada, uma lista pode ser aumentada. A sintaxe para se adicionar mais um elemento em uma lista é:

nome da lista`.append(`*novo elemento*`)`

O exemplo a seguir cria uma lista, depois aumenta em dois elementos.

```
>>> LL = [4,3,6,5]
>>> LL.append(8)
>>> LL.append(1)
>>> LL
[4, 3, 6, 5, 8, 1]
```

Tuplas

A sintaxe para se definir uma tupla é:

nome da tupla = (*elemento 1*, *elemento 2*, *elemento 3*, ...)

ou

nome da tupla = (*elemento*,)

A segunda tupla é chamada de *tupla unitária*, pois se trata de uma tupla com apenas um elemento. A vírgula que aparece depois do elemento é para indicar ao computador que, apesar de ter só um elemento, se trata de uma tupla.

Importante! Existe a necessidade dessa vírgula porque o símbolo dos parêntesis possui outros significados dentro da linguagem de programação e pode ser confundida com os parêntesis de uma expressão aritmética.

O exemplo a seguir cria uma tupla, chamada `y`, de números de 0 a 0,5 com incrementos de 0,1:

```
>>> y = (0, 0.1, 0.2, 0.3, 0.4, 0.5)
```

O exemplo a seguir cria uma tupla, chamada `semanas`, com os nomes dos dias da semana:

```
>>> semanas = ('Segunda-feira', 'Terça-feira', \
'Quarta-feira',  'Quinta-feira', 'Sexta-feira', \
'Sábado', 'Domingo')
```

Assim como as listas, os elementos de uma tupla não precisam ser do mesmo tipo, o exemplo a seguir cria uma lista, chamada `funcionario`, formada por elementos com tipos diferentes:

```
>>> funcionario = ('João', 35, 1320.18)
```

Apesar da semelhança em sua definição, listas e tuplas são diferentes em alguns aspectos:

- Uma vez criada, os elementos de uma lista podem ser modificados, enquanto

que os elementos de uma tupla não podem. Isso faz com que a lista seja muito mais flexível, e normalmente preferida, do que a tupla.

- Os recursos de memória necessários para armazenar uma lista são maiores do que de uma tupla, assim, quando não é necessário que os elementos sejam modificados depois da lista criada, é vantajoso se usar uma tupla.
- Quando se quer proteger uma lista para que não seja modificada depois de criada, como é o caso de chaves para bancos de dados, é necessário que se use uma tupla.

Existem algumas funções muito úteis para listas e tuplas, segue a descrição de algumas.

Comandos de Atribuição com Tuplas e Listas.

Quando o lado direito de um comando de atribuição é uma tupla ou lista, e no lado esquerdo encontra-se apenas uma variável, simplesmente a tupla, ou lista é criada com aquele nome. Existe porém, uma extensão a esse comando de atribuição que se aplica às tuplas e listas: quando no lado esquerdo do comando de atribuição são colocadas tantas variáveis quanto o número de componentes da tupla, ou lista, então cada elemento da tupla, ou lista é colocada na sua respectiva variável. O exemplo a seguir exemplifica o uso desse tipo de atribuição. Primeiramente cria-se uma tupla `t`. Em seguida usa-se a sintaxe estendida. Após o comando estendido `u` vale 1 e `v` vale 2.

```
>>> t = (1,2)
>>> t
(1, 2)
>>> u,v = (1,2)
>>> u
1
>>> v
2
```

Arrays

Arrays são a versão computacional para o conceito de vetor e matriz. A principal diferença entre uma lista ou tupla de uma *array* é o fato de que os componentes de uma *array* tem que ser todos do mesmo tipo, e com a restrição adicional de que sejam tipos numéricos.

Outra diferença é que as *arrays* não são um tipo nativo da linguagem, para utilizá-la é necessário importar o módulo `numpy`. Por ter um módulo específico para *arrays* eles são um tipo de dado bastante flexível.

Para se criar uma *array*, a sintaxe é:

nome da array `= array((`*elemento 1*`,` *elemento 2*`,` elemento 3`, …`
`),dtype=`*tipo de dados*`)`

O parâmetro `dtype` que pode valer `int`, `float` ou `complex` é opcional, caso omitido será usado o tipo de dado mais abrangente. Para não confundir com uma tupla, a palavra `Array` aparece antes da sequencia de números.

Os exemplos a seguir ilustram a criação de *arrays*:

```
>>> from numpy import array
>>> array((1,2,3))
array([1, 2, 3])
```

Sem o parâmetro dtype é criado um vetor de inteiros.

```
>>> array((1,2,3),dtype=float)
array([ 1.,  2.,  3.])
```

Nesse caso os números foram forçados a serem números reais.

```
>>> array((1,2,3),dtype=complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

Nesse caso são números complexos.

A manipulação de *arrays* será abordada com mais detalhes quando for tratado o assunto de matrizes e vetores.

Conversão de listas, tuplas e *arrays*.

Pode se criar uma tupla a partir de uma lista e vice e versa, a sintaxe para a conversão é:

nome da nova tupla = `tuple(`*nome da lista ou array*`)`

ou

nome da nova lista = `list(`*nome da tupla ou array*`)`

ou

nome da nova array = `array(`*nome da tupla ou lista*`, dtype=`*tipo*`)`

O exemplo a seguir ilustra essa conversão de tipos:

```
>>> x = (1,2,4)
>>> x.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute
'append'
```

Note que uma tupla não pode ser alterada uma vez criada.

```
>>> y = list(x)
>>> y.append(3)
>>> y
[1, 2, 4, 3]
```

A lista *y* foi criada a partir da tupla *x*. A lista *y* pode ser modificada.

A função `range`.

Essa é uma função que cria uma sequência de números inteiros. Essa sequência pode ser uma lista, se composta com a função `list`[3], ou tupla, se composta com a função `tuple`, ou uma *array*, se composta junto com a função `array`. Quando usada com um único parâmetro, retorna uma lista de inteiros de zero até o inteiro anterior ao número dado, o exemplo a seguir cria uma lista contendo os números de zero a nove.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
>>> array(range(10))
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Quando a função é chamada com dois parâmetros, a função gera uma lista começando do primeiro parâmetro e indo até o elemento imediatamente anterior ao segundo parâmetro. O exemplo a seguir cria uma lista de três a nove.

```
>>> list(range(3,10))
[3, 4, 5, 6, 7, 8, 9]
```

Quando a função é chamada com três parâmetros, o terceiro parâmetro indica o incremento, isto é, os números não aumentarão de um em um, mas sim seguirão o valor do incremento dado. O exemplo a seguir cria uma lista de 1 a 10 com incrementos de dois em dois.

```
>>> list(range(1,11,2))
[1, 3, 5, 7, 9]
```

`Varredura de Lista`

Em muitas situações é necessário se fazer uma varredura em todos os elementos de uma lista para gerar uma nova lista cujos valores são obtidos da primeira. A varredura de lista é usada, também, quando se deseja aplicar uma função a cada um dos elementos de uma lista. De forma geral sintaxe da varredura de lista é:

3 No *Python* 2 a função `range` pode ser usada sozinha, sem a necessidade de estar junto com `list` ou `tuple`, e vai gerar uma sempre uma lista, e não uma tupla.

nova lista = [*fórmula com* *x* `for` *x* `in` *sequência original*]

O resultado da função é uma nova lista com a fórmula dada aplicada a cada um dos elementos. O exemplo a seguir calcula a raiz quadrada de todos os inteiros de 1 a 10.

```
>>> import math
>>> x = range(1,11)
>>> [math.sqrt(z) for z in x]
[1.0, 1.4142135623730951, 1.7320508075688772, 2.0,
2.2360679774997898, 2.4494897427831779,
2.6457513110645907, 2.8284271247461903, 3.0,
3.1622776601683795]
```

Quando a função range for usada em uma varredura de lista, não é necessário que seja convertido para lista, tupla ou *array*.

O exemplo a seguir cria a função `sqr` e aplica a cada um dos elementos da lista de inteiros de 1 a 10.

```
>>> sqr = lambda x: x*x
>>> T = range(1,11)
>>> [sqr(x) for x in T]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

O exemplo anterior pode ser compactado para uma única linha, nesse caso não é necessário dar um nome à função (que era `sqr`) e nem dar um nome à lista (que era `L`).

```
>>> [x*x for x in range(1,11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

O exemplo a seguir gera uma lista de números reais igualmente espaçados entre dois números, sendo que são dados os valores inicial (*a*), final (*b*) e quantidade total de números (*n*). Para calcular a diferença (*delta*) entre dois pontos consecutivos calcula-se $delta = \frac{b-a}{n-1}$ em seguida gera-se uma lista *Li* com o índice de cada ponto, sendo que o índice do primeiro ponto é 0 e do último é *n*. Esse índice é calculado pela função `range`. Em seguida calcula-se cada um dos números pela fórmula $a + delta \times i$. Pela fórmula, quando *i* = 0 tem-se que *x* = *a*, e quando *x* = *n*-1 (último valor da lista *Li*) tem-se *x* = *b*. Finalmente a lista de números é mostrada na janela do terminal com o comando `x`.

```
>>> from __future__ import division
>>> a = 0
>>> b = 1
>>> n = 3
>>> delta = (b-a)/(n-1)
>>> x = [a+delta*i for i in range(n)]
>>> x
[0.0, 0.5, 1.0]
```

Índices para Tuplas, Listas, *Arrays* e *Strings*

Grandezas vetoriais são muito comuns nas ciências. Do ponto de vista computacional os vetores podem ser trabalhados como tuplas, listas ou *arrays*. O que caracteriza um vetor é a existência de componentes, que para uma lista seriam os elementos da lista. Uma diferença importante entre um vetor e uma lista é que para o primeiro todos os componentes tem que ser números reais, e para o segundo cada elemento pode ser de um tipo diferente, inclusive de tipo não numérico.

Por isso, para o fim de se representar vetores (e matrizes) o tipo estruturado mais adequado à a *array*. Entretanto em algumas situações podemos preferir um tipo nativo pois pode não haver garantia de que o módulo `numpy`, que define a *array*, esteja instalado no computador. Apesar de serem conceitualmente diferentes, pode-se usar tuplas e listas para representarem vetores.

Notação de índice

Uma vez criada uma tupla, lista ou *array*, muitas vezes é necessário fazer referência a apenas um item da dessa tupla ou lista. A forma de fazer referência a um elemento é através de um índice entre colchetes. O índice é baseado no zero, isto é, o primeiro elemento possui índice zero, o segundo possui índice um assim sucessivamente. O índice do último elemento é o tamanho da sequência menos um.

Pode-se usar também índices negativos. O último elemento é o elemento -1, o penúltimo -2, o antepenúltimo -3 e assim por diante.

O exemplo a seguir, em um terminal, cria uma tupla com três elementos. Em seguida mostra o primeiro elemento (de índice 0), o último (de índice 2) e o penúltimo usando um índice negativo (-2). Em seguida tenta mostrar um quarto elemento (de índice 3) e recebe uma mensagem de erro, uma vez que não existe um quarto elemento.

```
>>> u = (4, 2, 8)
>>> u[0]
4
>>> u[2]
8
>>> u[-2]
2
>>> u[3]
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    u[3]
IndexError: tuple index out of range
```

O exemplo a seguir cria uma tupla e uma lista de dois elementos cada. Em seguida mostra-se o primeiro elemento de cada uma. Depois altera o primeiro elemento da lista, e mostra a nova lista alterada. Por fim tenta alterar o primeiro elemento da tupla, e recebe uma mensagem de erro, pois uma vez criada os elementos de uma tupla não podem ser modificados.

```
>>> u = (3.2, 5.4)
>>> v = [3.3, 5.5]
>>> u[0]
3.2000000000000002
```

```
>>> v[0]
3.2999999999999998
>>> v[0]=1.5
>>> v
[1.5, 5.5]
>>> u[0]=1.5
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    u[0]=1.5
TypeError: 'tuple' object does not support item
assignment
```

A exemplo das sequências, um caractere de uma *string* pode ser endereçado usando a mesma notação de índices. Pode-se pensar na *string* como uma sequência de caracteres.

O exemplo a seguir cria uma *string* e faz mostra o primeiro, segundo e décimo primeiro caractere. Em seguida tenta alterar o primeiro caractere e recebe uma mensagem de erro. A exemplo das tuplas, as strings não podem ser alteradas.

```
>>> s = 'testando...'
>>> s[0]
't'
>>> s[1]
'e'
>>> s[10]
'.'
>>> s[0] = 'b'
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    s[0] = 'b'
TypeError: 'str' object does not support item
assignment
```

Notação de fatias

Muitas vezes é necessário pegar apenas uma parte de uma tupla, lista ou string, nesse caso usa-se o símbolo dois pontos (`:`) para separar o índice inicial e o índice final. A forma

índice inicial **:** *índice final*

É lida como “de” índice inicial “até” índice final. A omissão do índice inicial indica “do início”, e a omissão do índice final indica “até o fim”.

Uma imagem que ajuda a entender os índices é pensar que os espaços entre os elementos é que são numerados. Os índices da string `'testando...'` poderiam ser:

```
| t | e | s | t | a | n | d | o | . | . | . |
0   1   2   3   4   5   6   7   8   9  10  11
```

```
-11 -10 - 9  -8  -7  -6  -5  -4  -3  -2  -1
```

Assim algumas fatias poderiam ser:

```
>>> s[1:3]
'es'
>>> s[:3]
'tes'
>>> s[3:]
'tando...'
>>> s[3:-1]
'tando..'
```

Para a tupla (4, 6, 8, 10) os índices são:

```
tupla:              | 4 | 6 | 8 | 10 |
índices:            0   1   2   3    4
índices negativos: -4  -3  -2  -1
```

Algumas fatias possíveis:

```
>>> u = (4,6,8,10)
>>> u[1:3]
(6, 8)
>>> u[:3]
(4, 6, 8)
>>> u[:-2]
(4, 6, 8)
>>> u[3:]
(10,)
```

Na última linha a vírgula depois do dez indica que é uma *tupla* unitária e não apenas o *número* 10.

Dicionários

Dados do tipo Dicionário são parecidos com listas e tuplas, exceto que cada elemento é formado por um par de valores. O primeiro subelemento é chamada de *key* (chave) e é usada para fazer referência ao elemento. O segundo subelemento é chamado de *value* (valor) e é o valor que está armazenado no elemento. Pode-se pensar no dicionário como uma lista de variáveis, em que a chave é o nome da variável de o valor o seu conteúdo.

Outra coisa de diferencia o dicionário de listas e tuplas é que os elementos não são ordenados, a ordem que são guardados na memória da máquina é decidida pelo próprio computador, sendo que o programador não tem controle sobre isso.

A maneira de se criar um dicionário é:

nome do dicionário = { *chave1*:*valor1*, *chave2*:*valor2*, ... }

Para fazer referência um valor de um dicionário usa-se a notação com abre e fecha

colchetes, onde se usa o valor da chave especificar qual o elemento a ser usado. Para obter um valor de um dicionário usa-se

nome do dicionário`[`*chave*`]`

para modificar um valor usa-se

nome do dicionário`[`*chave*`]` = *novo valor*

O exemplo a seguir cria um dicionário, chamado `funcionario`, com alguns dados. Em seguida altera o campo idade para 40.

```
>>> funcionario = {'nome':'Renato', 'idade':39,  \
... 'salario':1230.18}
>>> funcionario['idade'] =40
>>> funcionario
{'idade': 40, 'salario': 1230.1800000000001, 'nome':
'Renato'}
```

Raramente dados estruturados na forma de um dicionário são usados em métodos numéricos, mas essa estrutura de dados com frequência é usada por algumas funções internas da própria linguagem de programação, como é o caso da função `vars`, que fornece um dicionário de todas as variáveis do sistema.

Funções úteis dentro do Terminal

Algumas funções são usadas para realizar tarefas e buscar informações do próprio terminal, a seguir estão relacionados alguns desses comandos:

Função	Efeito
`quit()` ou `exit()`	Sai do Terminal do Interpretador
`vars()`	Retorna um dicionário com todas as variáveis e seus valores.
Para usar as funções abaixo é necessário primeiro importar o módulo `os` com o comando `>>> import os`	
`os.getcwd()`	Retorna uma string com o nome da pasta atual. A palavra `getcwd` é uma abreviação da frase em inglês "*get current working directory*", isto é "pegue o diretório (pasta) atual"
`os.listdir(`*pasta*`)`	Retorna uma lista com os nomes dos arquivos da pasta especificada. Para ficar mais fácil de ler é melhor digitar o comando `>>> '\n'.join(os.listdir(`*pasta*`))`
`os.chdir(`*nova pasta*`)`	Muda para uma nova pasta. (abreviação de "*change directory*" que significa "mude o diretório (pasta)"

Programação

Diferentemente do terminal, em que cada comando ou fórmula é digitada e executada, assim que a tecla “enter” é pressionada, na programação digitam-se vários comandos que depois serão executados sequencialmente. A vantagem desse processo é que se pode automatizar todo o processo de cálculo e, depois, executá-lo várias vezes para dados, ou funções, diferentes. Assim, por exemplo, pode-se estabelecer uma sequencia de comandos que calcularão a área sob uma função (num intervalo), digitar esses comandos e, depois, executar essa sequência para diferentes funções ou diferentes intervalos.

O que é preciso para começar a programar?

1. **O Interpretador do *Python* instalado.** Se o terminal do *Python* está funcionando, essa etapa já está cumprida. Caso não esteja é necessário baixar o *Python* e instalá-lo;
2. **Um editor de texto não formatado como o "Bloco de Notas" do *Windows* ou o "gedit" do Gnome (Linux).** Editores de texto formatados como o *Word* e o BrOffice *Writer* não podem ser usados porque eles acrescentam no texto marcações internas que o interpretador não reconhece.

Existem ambientes mais sofisticados para o desenvolvimento de programas em *Python*, entretanto o conhecimento desses ambientes foge ao objetivo desse texto.

O Primeiro Programa: `print` e o "Alo Mundo"

O pequeno programa a seguir escreve a frase "Alo Mundo" na janela do terminal. O comando para se escrever alguma coisa na tela é a função

```
print('texto')
```

onde o texto que segue o comando é o texto que aparecerá na tela. Se o texto for uma *string*, essa *string* será exibida na tela, se for uma fórmula, o resultado da fórmula aparecerá.

O comando `print` é na realidade uma função que não retorna valor, mas que tem como “efeito colateral” a escrita de um texto na janela do terminal.

O programa `ola.py` terá apenas uma linha de código fonte, com o comando

```
print('Alo Mundo')
```

Para escrever, e executá-lo:

1. Abra o editor de texto e digite o comando

```
print('Alo Mundo')
```

Todo o texto a ser exibido (a frase: Alo Mundo) deverá estar entre aspas pois se trata de uma *string*. Caso não tenha aspas o texto seria interpretado como uma fórmula, e o interpretador do tentaria fazer a conta antes de mostrar o resultado.

2. Salve o texto do programa num arquivo com extensão ".py", como por exemplo "`ola.py`".

3. Entre no Terminal do Python e digite

```
>>> exec(open('ola.py').read())
```
[4]

Deverá aparecer a frase "Alo Mundo" no Terminal do Python.

Alternativamente é possível executar o programa diretamente de um terminal do próprio sistema operacional, sem ter que entrar no Terminal do *Python*. Basta digitar, a partir do terminal

```
c:\ python ola.py
```

Exercício:

3. Faça um programa que mostre o seu nome na tela.

Interação com o usuário, acentos e comentários (#)

Comentários

Outro elemento importante em um programa são os *comentários*. Tudo que aparece depois do caractere # será ignorado pelo interpretador. Nesse local normalmente são colocadas informações importantes para o programador conseguir entender o código que foi escrito.

Entrada de Dados: `input(...) e raw_input(...)`

Muitas vezes deseja-se que dados sejam fornecidos a um programa durante a sua execução, e esses dados são armazenados em variáveis. Para solicitar informações durante a execução existem as funções `input e raw_input`. Estas funções interrompem a execução do programa e espera que o usuário digite alguma coisa e depois pressione a tecla <enter>.

Ao digitar números, o usuário pode fornecer um número sem casas decimais quando o valor esperado é um número real, ou, por engano, colocar casas decimais em um número inteiro. Por isso, no caso de números, é importante fazer uma conversão de tipo antes de armazenar o número. Assim existem basicamente três formas de se solicitar um dado do usuário:

Para solicitar um número inteiro:

```
nome da variável = int(input('Texto a ser mostrado '))
```

Note a composição da função `input` com a função `int`. Isso garante que o resultado final armazenado na variável seja um número inteiro descartando qualquer casa decimal que porventura seja digitada.

Para solicitar um número real:

4 Essa sintaxe parece a princípio bastante confusa, entretanto é a única que é compatível entre as duas versões. Na antiga versão (a versão 2.x) existe uma função mais simples que é `>>> execfile('ola.py')`.

nome da variável = float(input('*Texto a ser mostrado* '))

Note a composição da função input com a função float. Isso garante que o resultado final que será armazenado na variável seja um número real, acrescentando zeros nas casas decimais caso o usuário não tenha digitado nenhuma casa decimal.

Para solicitar uma *string*:

nome da variável = raw_input('*Texto a ser mostrado* ')[5]

Saída de Dados: `print`

O comando de saída de dados para a tela é a função print, conforme usado no exemplo anterior. Para manter a compatibilidade entre as versões principais, é importante importar o recurso print_function do módulo __future__. Assim, sempre que se for usar a função print é importante colocar no início do programa a linha

```
from __future__ import print_function
```

Uma das variações da sintaxe da função é a que permite que várias *strings* sejam escritas. De forma mais geral a função print é

print(*string 1, string 2, ..., string n*)

Assim cada *string* aparecerá separada por um espaço em branco.

Para a função print existem alguns códigos com significado especial chamados de caracteres de escape, e são indicados pelo caractere "\" (barra invertida). Um deles que é o \n (barra invertida seguida do n). Isso é interpretado como "mude de linha", uma espécie de abreviação de *new line*. Ao inserir esse código no texto a ser mostrado, pode-se inserir linhas em branco no texto tornando-o mais legível na tela.

A relação abaixo mostra alguns dos caracteres de escape úteis para o comando print:

\n - quebra a linha
\" - mostra o caractere aspas
\' - mostra o caractere apóstrofo
\\ - mostra a barra invertida
\t - caractere de tabulação

Por exemplo, para colocar um apóstrofo, ou aspa, no interior de uma *string* é necessário indicar que aquele apóstrofo ou aspa não é um sinal de que a *string* acabou. Para indicar isso usa-se a barra invertida (\) andes do apóstrofo ou aspa. Para indicar uma situação em a barra invertida não é um caractere de escape, usa-se a barra invertida duas vezes (\\). O exemplo a seguir mostra uma *string* que possui um apóstrofo no meio.

5 Essa função não existe na versão 3.x, ela deverá ser substituída por input(*Texto a ser mostrado*)

```
>>> print('don\'t stop')
don't stop
```

Quando se deseja usar letras acentuadas, devido à diferença de codificação de caracteres nos diferentes sistemas operacionais (*Windows*, Linux, etc.), é preciso indicar ao interpretador qual a codificação que está sendo usada. Essa informação, como não faz parte do programa, é feita através de uma linha de comentário. Deve-se colocar na *primeira ou segunda linha do programa* um comentário com a palavra '`coding:`' e o tipo de codificação. Normalmente a codificação é '`cp1252`' para *Windows* e '`utf-8`' para Linux. Assim a linha de comentário fica:

para *Windows*

```
# coding: cp1252
```

e para Linux

```
# coding: utf-8
```

O exemplo a seguir solicita do usuário que entre com o nome, em seguida faz uma saudação personalizada. O programa se constitui de apenas duas linhas executáveis, a que solicita o nome e a que escreve a saudação. As demais linhas são comentários para configurar a codificação de caracteres e explicar alguma coisa de cada linha.

```
1           # programa alo2.py
2           # coding: cp1252
3           # Segundo exemplo do programa alô.
4           from __future__ import print_function
5
6           # tipo a ser solicitado: string.
7           nome = raw_input('Entre com o nome -> ')
8
9           # três strings são escritas
10          print('\n\nalô',nome,'\n\n')
```

`print` com `string` formatada

Com frequência, em um comando `print`, é necessário misturar em um único texto os valores de uma variável com o texto explicativo. Existe uma construção que permite fazer essa mistura com certa facilidade: é o método `format`.

A forma mais simples de usar essa construção é de marcar a posição onde o conteúdo da variável (ou resultado da expressão) aparecerá, com chaves que envolvem o número que representa a variável que será colocada. Além disso, dentro do método `format` da `string` listam-se quais as variáveis que serão utilizadas. O exemplo abaixo ilustra essa forma:

```
1           # programa format1.py
2           # coding: utf-8
```

```
3           from __future__ import print_function
4
5           n = 3
6           x = 1.0/n
7           print('O valor de 1/{0} é {1}.'.format(n,x))
```

O resultado é:

```
O valor de 1/3 é 0.333333333333.
```

Nesse exemplo o método `format` possui dois argumentos: `n` (cujo índice é *zero*) e `x` (cujo índice é *um*). Dentro da `string` a marcação `{0}` é substituída pelo valor de `n` e a marcação `{1}` é substituída pelo valor de `x`.

As marcações {*número*} podem aparecer em qualquer ordem, e qualquer número de vezes. Os valores das variáveis também podem ser *strings*, e não apenas números. O exemplo a seguir ilustra esses fatos:

```
1           # programa format2.py
2           # coding: utf-8
3
4           n = 2
5           s = 'dois'
6           print('{0} é o mesmo que {1}'.format(n,s))
7           print('{1} é o mesmo que {0}'.format(n,s))
```

O resultado do programa acima é:

```
2 é o mesmo que dois
dois é o mesmo que 2
```

Além de mesclar o conteúdo de uma variável, é possível, também, incluir a especificação da largura que se deseja reservar para um número, assim como o número de casas decimais que se deseja mostrar. Para especificar a formatação com mais detalhes usam-se códigos dentro da marcação dos parêntesis, esses códigos são colocados depois do sinal de : (dois pontos).

Para números inteiros usa-se o código:

:*n*d

onde *n* é o número mínimo de espaços reservados para o número. O número ficará alinhado com a margem direita do espaço reservado ao número.

Para números reais, pode-se fixar o número de casas decimais a ser mostrado usando o código

:.*n*f

onde *n* é o número de casas decimais para o arredondamento. Para fixar a largura mínima e o número de casas decimais tem-se:

:*m*.*n*f

onde *m* é o número mínimo de caracteres que o número usará, e *n* é o número de casas decimais.

Pode-se forçar com que os números sejam escritos em potência de dez usando o código e. Além da potência de dez pode-se especificar o número de casas decimais da mantissa. O exemplo a seguir ilustra cada um dos casos.

```
1           # programa format3.py
2           # coding: utf-8
3           import math
4           print('Inteiro sem formatação -> {0}'.format(3))
5           print('Inteiro com formatação ":5d" ->
            {0:5d}'.format(3))
6           print('Real sem fomatação -> {0}'.format(math.pi))
7           print('Real com código ":.3f" ->
            {0:.3f}'.format(math.pi))
8           print('Real com código ":10.3f" ->
            {0:10.3f}'.format(math.pi))
9           print('Real com código ":e" ->
            {0:e}'.format(math.pi))
10          print('Real com código ":.3e" ->
            {0:.3e}'.format(math.pi))
```

A saída desse programa é dado abaixo:

```
Inteiro sem formatação -> 3
Inteiro com formatação ":5d" ->     3
Real sem fomatação -> 3.14159265359
Real com código ":.3f" -> 3.142
Real com código ":10.3f" ->      3.142
Real com código ":e" -> 3.141593e+00
Real com código ":.3e" -> 3.142e+00
```

Mantendo o resultado do `print` na mesma linha

Entre duas chamadas à função `print` consecutivas, sempre é inserida uma nova linha. Em algumas situações é necessário que o resultado do comando `print` seguinte fique na mesma linha do resultado anterior. Esse efeito é conseguido através do parâmetro opcional `end`. Esse parâmetro é, por padrão, igual a `'\n'`, que instrui a mudar de linha. Esse valor pode ser modificado para um espaço em branco, ou até para uma *string* vazia. O exemplo a seguir ilustra o uso do parâmetro `end`.

```
1           # programa printEnd.py
2           # coding: utf-8
3           from __future__ import print_function
4
5           print('\n\nabc')
6           print('def')
7           print('ghi', end=' ')
8           print('jkl', end='')
9           print('mno\n\n')
```

A saída do programa `printEnd.py` forneceu o resultado abaixo:

```
abc
def
ghi jklmno
```

Após o primeiro e segundo comandos `print`, houve a introdução de uma nova linha. Já após o terceiro `print`, o qual foi adicionado o parâmetro `end=' '` foi introduzido apenas um espaço em branco, e o resultado da quarta linha de comando permaneceu na terceira linha. Como o quarto `print` possui o parâmetro foi uma *string* vazia (`end=''`), o resultado do quinto `print` também foi na terceira linha, e sem nenhum tipo de caractere entre as duas saídas.

Exercícios:

Importante: Os exercícios são um elemento importante no aprendizado da Simulação Matemática. Muitos desses exercícios possuem soluções já prontas na internet (e futuramente nesse próprio texto). Entretanto evite a tentação de buscar a solução imediatamente. Tente fazer cada exercício por conta própria, busque uma solução já pronta apenas se tentou fazer, e não conseguiu por mais de **vinte** minutos.

Uma vez que conseguiu fazer um programa, faça alterações para verificar se há formas de melhorá-lo.

4. Faça um programa que mostre o seu nome na tela. Deverá haver duas linhas em branco antes do nome, e duas linhas em branco depois do nome.
5. Faça um programa que peça um número e então mostre a mensagem *O número informado foi [número]*.
6. Faça um programa que peça dois números e imprima a soma.
7. Faça um programa que peça as 4 notas bimestrais e mostre a média.
8. Faça um programa que converta metros para centímetros.
9. Faça um programa que peça o raio de um círculo, calcule e mostre sua área.
10. Faça um programa que pergunte quanto alguém ganha por mês e o número de horas trabalhadas no mês. Calcule e mostre o salário por hora no referido mês.
11. Faça um Programa que peça a temperatura em graus *Fahrenheit*, transforme e mostre a temperatura em graus Celsius, mostrando o resultado até 4 casas decimais.

$$C = \frac{5(F-32)}{9}$$

12. Faça um Programa que peça a temperatura em graus Celsius, transforme e mostre em graus *Fahrenheit*, mostrando o resultado até 4 casas decimais.

13. Tendo como dados de entrada a altura de uma pessoa, construa um programa que calcule seu peso ideal, utilizando as seguintes fórmulas:

    Para homens: 72.7 *h* - 58
    Para mulheres: 62.1 *h* – 44.7
    onde *h* é a altura informada

14. Faça um Programa que peça 2 números inteiros e um número real. Calcule e mostre:

    a) o produto do dobro do primeiro com metade do segundo .
    b) a soma do triplo do primeiro com o terceiro.
    c) o terceiro elevado ao cubo.

15. Faça um programa que pergunte quanto alguém ganha no mês de salário bruto. Calcule e mostre o total do seu salário líquido no referido mês, sabendo-se que são descontados 11% para o Imposto de Renda, 8% para o INSS e 5% para o sindicato. A saída do programa deve escrever o seguinte texto:

```
+ Salário Bruto: R$ valor
- IR (11%): R$ valor
- INSS (8%): R$ valor
- Sindicato ( 5%): R$ valor
= Salário Liquido: R$ valor
```

    Obs.: Salário Bruto - Descontos = Salário Líquido.

16. Faça um Programa que leia três números e mostre o maior deles. (Obs. Pode-se usar a função predefinida `max`, da página 18)

17. Faça um Programa que leia três números e mostre o maior e o menor deles. (Obs. Pode-se usar as funções predefinidas `max` e `min` da página 18)

18. Faça um programa que pergunte o preço de três produtos e informe qual produto você deve comprar, sabendo que a decisão é sempre pelo mais barato.

19. Faça um programa que leia três números e mostre-os em ordem decrescente. (Dica: Guarde os números como uma lista usando o método `append`, aplique o método `sort` na lista)

20. Faça um programa que leia 5 números e informe o maior número. (Dica: Guarde os números como uma lista)

Declaração Condicional Simples

Um elemento importante na programação é a habilidade do computador fazer

coisas diferentes dependendo das informações que o usuário fornecer, ou dependendo dos resultados das contas. Nesse sentido o `if` é uma declaração que instrui o computador a executar um bloco de comandos *apenas se uma dada condição é verdadeira*. A forma geral do comando `if` é:

```
if condição:
    comando 1
    comando 2
    etc
    comando n
próximo comando não recuado
```

A condição a ser testada normalmente é uma expressão lógica cujo resultado é verdadeiro ou falso. Se o resultado da condição, que fica entre a palavra `if` e os dois pontos (`:`), for verdadeira, os comandos recuados: comando 1, comando 2, etc. até comando *n* serão executados, e em seguida a execução passa para o próximo comando não recuado. Por outro lado, se o resultado for falso, a execução passa diretamente para o próximo comando não recuado, pulando todos os comandos que estão recuados.

Duas coisas importantes e abrangentes aparecem no comando `if`: a expressão lógica, e o recuo.

Expressão Lógica

Expressões lógicas são fórmulas de comparação do tipo “igual a”, “maior que”, “diferente de” cujo resultado é falso ou verdadeiro. Para fazer as comparações usam-se os símbolos:

Símbolo	Descrição
<	é menor que
<=	é menor ou igual a
>	é maior que
>=	é maior ou igual a
==	é igual a
!= ou <>	é diferente de

Nos exemplos a seguir, o conteúdo de duas variáveis x e y são comparadas:

```
>>> x = 3.5
>>> y = 1.2
```

```
>>> y < x
True
>>> y == x
False
>>> y <> x
True
```

Cuidado! - É muito comum ocorrer uma confusão entre o sinal de atribuição "=" e a comparação "==" entretanto eles são **muito** diferentes. A expressão `x=3` é uma instrução que manda guardar o número 3 na variável `x`. Enquanto que a expressão `x==3` é uma comparação de `x` com o número 3, que será `True` se `x` guardar o número 3 e `False` se `x` guardar qualquer outro número.

Importante! - Além de verdadeiro e falso, o `if` interpreta também outros tipos de resultados como se fossem verdadeiro ou falso. São interpretados como *falso*: o número zero, e uma *string*, tupla, lista ou dicionário vazio. E como *verdadeiro*: qualquer valor diferente de zero, e *string*, tupla, lista ou dicionário não vazio.

O operador `in` (pertence)

É muito comum a necessidade de se verificar se um elemento pertence a um conjunto. Nesse sentido dados do tipo "sequência", que são lista, tupla, *array* e *string* podem ser vistas como conjuntos em que se deseja saber se algum elemento pertence a ele, ou não. Para essa verificação usa-se a seguinte sintaxe:

elemento `in` *sequência*

O resultado dessa expressão será `True` caso o elemento exista na sequencia, e `False` caso ele não exista. O programa a seguir ilustra o uso do operador in.

```
1           # programa in.py
2           # coding: utf-8
3           from numpy import array
4
5           T = (1,2,3)
6           L = [4,5,6]
7           A = array((7,8,9),dtype=float)
8           S = '8910'
9
10          x = int(input('Entre com um número -> '))
11
12          if x in T:
13            print('\nO número {0} pertence à tupla
            T'.format(x))
14
15          if x in L:
16            print('\nO número {0} pertence à lista
```

```
            L'.format(x))
17
18          if x in A:
19            print('\nO número {0} pertence à array
            A'.format(x))
20
21          x = str(x)
22          if x in S:
23            print('\nO texto \'{0}\' pertence à string
            S'.format(x))
```

Operações com Expressões Lógicas

As expressões de comparação podem ser combinadas pelas operações lógicas `and` (e), `or` (ou não exclusivo) e `not` (inversão). As tabelas abaixo listam a forma com que cada operação lógica combina as expressões:

x	***y***	***x* and *y***	***x* or *y***	**not** ***x***
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

O exemplo a seguir mostra como duas comparações são combinadas

```
>>> x = 3.5
>>> y = 1.2
>>> (y < x) and (x > 3)
True
```

O Recuo

O recuo é um elemento fundamental de muitos comandos, pois faz parte de sua sintaxe. Os comandos recuados formam um grupo, comumente chamado de bloco de comandos. Os comandos desse bloco tem a seguinte relação: ou *todos* são executados, ou *nenhum deles* é. No caso do comando `if` quem determina se o bloco vai ser, ou não, executado é a condição que foi colocada no comando.

O recuo pode ser de qualquer quantidade, um ou mais espaços em branco. Recomenda-se que se use quatro espaços em branco, e que *não* se use a tabulação para fazer esse recuo.

Dentro de um bloco de comandos pode haver algum deles que, em si, exige um outro recuo.

O programa a seguir pede o nome e o sexo. Baseando-se na resposta ele emite um

cumprimento personalizado para o nome e o gênero.

```
1           # programa alo3.py
2           from __future__ import print_function
3
4           nome = raw_input('\n\nnome -> ')
5           sexo = raw_input('sexo (m ou f) -> ')
6           s = 'Oi senhor'
7           if sexo=='f':
8               s += 'a'
9           print(s, nome, '\n')
```

A linha recuada, depois do `if`, só será executada se o usuário digitar a letra `f` na pergunta sobre o sexo.

O `If` com duas opções que se excluem: a cláusula `else`

Usa-se o `if` com `else` na situação em que um bloco de comandos vai ser executado quando uma condição é verdadeira, e *um outro* bloco de comandos vai ser executado quando a condição for falsa. A estrutura geral do `if` com `else` é:

`if` *condição*`:`
    comando para verdadeiro 1
    comando para verdadeiro 2
    etc
    comando para verdadeiro n
`else:`
    comando para falso 1
    comando para falso 2
    etc
    comando para falso m
próximo comando não recuado

Nesse caso apenas um dos blocos de comandos vai ser executado.

O exemplo a seguir pede um número e verifica se ele é par ou impar. Quando o resto da divisão do número dado por 2 for zero, o número é par, caso contrário (quando o resto é um) o número é impar.

```
1           # programa par.py
2           # coding: utf-8
3           from __future__ import print_function
4
5           n = int(input('Número natural -> '))
6           if n%2 == 0:
7               s = 'par'
8           else:
```

```
9                s = 'impar'
10           print('O número {0} é {1}'.format(n,s))
```

Exercício:

21. Um pescador comprou um computador para controlar o rendimento diário de seu trabalho. Toda vez que ele traz um peso de peixes maior que o estabelecido pelo regulamento de pesca, que é de 50 quilos, ele deve pagar uma multa de R$ 4,00 por quilo excedente. Faça um programa que leia o peso dos peixes e verifique se há excesso. Se houver, guardar na variável `excesso` e na variável `multa` o valor da multa que João deverá pagar. Caso contrário mostrar tais variáveis com o conteúdo ZERO.

Atribuição com o truque do `Andor`

Pode-se fazer uma atribuição que testa uma condição, se for verdadeira atribui um valor, se for falsa atribui outro. Esse comportamento é análogo ao `if` com `else`, e pode ser escrita em uma única linha. É um truque muito útil quando combinada com a função lambda, pois permite que se escrevam funções que são definidas por partes. A sintaxe do truque do Andor é:

resultado `=` *condição* `and` *valor se verdadeiro* `or` *valor se falso*

O exemplo abaixo ilustra o uso da atribuição com o truque do Andor:

```
>>> x = 2
>>> y = x>0 and 1 or -1
>>> y
1
>>> x = -2
>>> y = x>0 and 1 or -1
>>> y
-1
```

Pode-se observar que quando *x* era positivo, o valor de *y* foi 1. Quando *x* passou a ser negativo, o valor de *y* foi -1.

Esse truque se vale do fato de que as operações lógica `and` e `or` quando aplicadas a valores numéricos, não retornam `True` ou `False`, mas sim o valor de um dos operandos.

Outra característica das expressões lógicas, é que elas podem ser incompletas, isto é, as operações são feitas da esquerda para a direita da expressão. Se o resultado da expressão lógica puder ser obtida antes do final da sua avaliação, o cálculo da expressão não é executada até o fim. Assim na expressão

`True and True or` *qualquer coisa*

o resultado independe da última parcela. Se o primeiro operando é “verdadeiro”, e o segundo é “verdadeiro”, o resultado da primeira parte da expressão é “verdadeiro” independentemente do que vem depois da operação `or`. Por isso a operação `or` não será executada.

Em particular a operação `and`, quando o primeiro argumento é `True`, retorna o

segundo argumento, e quando o primeiro argumento é `False`, ele retorna `False`. Conforme visto no exemplo abaixo.

```
>>> True and 2
2
>>> False and 2
False
```

Diferentemente, a operação or, quando o primeiro argumento é "verdadeiro", ou alguma coisa que possa ser interpretada como "verdadeiro", retorna "verdadeiro". Quando o primeiro operando é "falso", ele retorna o outro operando. Conforme pode ser visto no exemplo abaixo.

```
>>> True or 2
True
>>> False or 2
2
```

Entretanto há um perigo nessa expressão. Se o segundo operando do and for alguma coisa que possa ser interpretada como "falso", como por exemplo o número zero, ou string, tupla, lista ou dicionário vazio, o truque não funcionará. Para evitar esse tipo de problema é sempre aconselhável colocar os valores retornados como uma tupla, ou lista unitária. Dessa forma o resultado nunca será alguma coisa que possa ser interpretada como "falsa". Para obter o resultado final basta tomar o primeiro elemento da tupla, ou lista. Assim uma construção mais garantida é:

resultado = (*condição* and (*valor se verdadeiro*,) or (*valor se falso*,))[0]

ou

resultado = (*condição* and [*valor se verdadeiro*] or [*valor se falso*][0]

A função declaração a seguir cria uma função, $q(x)$ que é zero quando x é negativo, e x^2 quando x é positivo.

```
>>> q = lambda x:(x<0 and (0,) or (x**2,))[0]
>>> q(-4)
0
>>> q(3)
9
```

Testes com várias condições: a cláusula `elif`

Tanto o `if` simples, como o `if` com cláusula `else` testam apenas uma condição. Entretanto existem situações em que se necessita testar várias condições, ou casos, onde apenas uma é verdadeira, e apenas esse bloco será executado. Essa situação pode ser trabalhada com vários comandos `if`, um para cada teste. Contudo há uma variação do `if`

própria para isso, que é a que se segue:

```
if condição 1:
    comando se condição 1 for verdadeira 1
    ...
    comando se condição 1 for verdadeira n1
elif condição 2:
    comando se condição 2 for verdadeira 1
    ...
    comando se condição 2 for verdadeira n2
elif condição 3:
    comando se condição 3 for verdadeira 1
    ...
    comando se condição 3 for verdadeira n3
else:
    comando se todas as condições anteriores forem falsas 1
    ...
    comando se todas as condições anteriores forem falsas n
próximo comando depois do if
```

Caso a *condição 1* seja verdadeira, **apenas** o bloco de comandos que está recuado depois do teste da *condição 1* é que será executado. Se a condição 1 for falsa, a condição 2 será testada. Caso a condição 2 seja verdadeira, **apenas** o bloco de comandos que está recuado depois do teste da *condição 2* é que será executado. Assim sucessivamente até encontrar uma condição verdadeira, ou que todas as condições sejam falsas. Caso todas as condições forem falsas o bloco de comandos depois da clausula `else` sera executado.

Podem existir tantos testes `elif` quanto sejam necessários, e a clausula `else` é opcional

O exemplo a seguir pede um número, que representa uma temperatura, do teclado, e classifica em três situações: muito fria, amena e muito quente. A temperatura será considerada muito fria se for menor que 10 graus, amena se estiver de 10 a 27 graus e muto quente se estiver acima de 27 graus.

```
1           # programa temperatura.py
2           x = float(input('Entre com uma temperatura -> '))
3
4           if x < 10:
5               print('Muito fria')
6           elif x <= 27:
7               print('Amena')
8           else:
9               print('Muito quente')
```

Exercícios.

22. Faça um Programa que pergunte em que turno o usuário estuda. Peça para digitar M para matutino, V para vespertino ou N para noturno. Imprima a mensagem "Bom Dia!", "Boa Tarde!" ou "Boa Noite!" ou "Valor Inválido!", conforme o caso.

23. Uma empresa resolveu dar um aumento de salário aos seus colaboradores. Faça um programa que recebe o salário de um colaborador e calcula o reajuste segundo o seguinte critério, baseado no salário atual:

    salários até R$ 280,00 (incluindo) : aumento de 20%
    salários entre R$ 280,00 e R$ 700,00 : aumento de 15%
    salários entre R$ 700,00 e R$ 1500,00 : aumento de 10%
    salários de R$ 1500,00 em diante : aumento de 5%

    Após o aumento ser realizado, informe na tela:
    o salário antes do reajuste;
    o percentual de aumento aplicado;
    o valor do aumento;
    o novo salário, após o aumento.

24. Tendo como dados de entrada a altura e o gênero de uma pessoa, construa um programa que calcule seu peso ideal, utilizando as seguintes fórmulas:

    a) Para homens: 72.7*h - 58
    b) Para mulheres: 62.1*h - 44.7 (h = altura)
    c) Em seguida peça o peso da pessoa e informe se ela está dentro, acima ou abaixo do peso.

25. Faça um programa que peça dois números e imprima o maior deles (sem usar a função `max`).

26. Faça um programa que peça um valor e mostre na tela se o valor é positivo ou negativo.

27. Faça um programa que verifique se uma letra digitada é "F" ou "M". Conforme a letra escrever: F - Feminino, M - Masculino, Sexo Inválido.

28. Faça um programa que verifique se uma letra digitada é vogal ou consoante (obs. É vantajoso usar o operador `in` da página 48).

29. Faça um programa para a leitura de duas notas parciais de um aluno. O programa deve calcular a média alcançada por aluno e apresentar:

    A mensagem "Aprovado", se a média alcançada for maior ou igual a sete;
    A mensagem "Reprovado", se a média for menor do que sete;
    A mensagem "Aprovado com Distinção", se a média for igual a dez.

Depuração de programas

Chama-se de *depuração* ao ato de procurar (e corrigir) erros no programa. Os erros normalmente são de dois tipos: de sintaxe e lógica.

Os erros de *sintaxe* ocorrem quando se escreve alguma coisa que foge às regras da linguagem e não pode ser interpretada. Problemas de sintaxe comuns são: digitação errada de nome de variáveis e de funções, falta se sinais de pontuação, principalmente os dois pontos depois de comandos como o `if` que já foi visto. Nesses casos o interpretador costuma apontar o erro.

Entretanto erros de *lógica* (ou de execução) são mais sutis. Esses erros ocorrem por vários motivos, desde erros de digitação, como quando se digita o número `1`, quando

deveria digitar a letra `l` (L minúsculo) até erros conceituais, onde o computador está fazendo o que foi instruído, mas as instruções estão erradas.

Para detectar erros de lógica existem estratégias simples e básicas que podem ser usadas:

- Executar o programa com dados cujos resultados são conhecidos. Essa estratégia consiste em fornecer os dados de entrada para casos simples em que o resultado é conhecido;
- Exclusão de linhas por comentários. Essa estratégia consiste na colocação do símbolo de comentário (#) no início das linhas onde se suspeita que os problemas estejam surgindo. Dessa forma pode-se isolar a linha que está com problemas;
- Escrever no terminal os valores das variáveis suspeitas e textos de teste. Pode-se colocar a função `print` dentro de blocos de `if` para certificar se o programa está passando por onde deveria, ou escrever os valores intermediários das variáveis suspeitas de estar produzindo o erro.

Repetição de comandos com o `while`

Um comando, ou grupo de comandos pode ser repetido enquanto uma determinada condição for verdadeira. Uma vez que essa condição é falsa, as repetições cessam. Isso é conseguido usando o comando `while`, que possui a seguinte forma geral:

`while` *condição de continuidade*`:`
    comando 1
    comando 2
    etc.
    Comando n
próximo comando depois do bloco do while

A condição de continuidade é calculada. Se ela for verdadeira todo o bloco de comandos que estão recuados será executado. Ao final a condição de continuidade é calculada novamente, se ela continuar verdadeira o bloco de comandos será executado novamente. Esse processo se repete até que a condição de continuidade calculada se torne falsa. Assim o próximo comando que não está recuado é executado e a execução do restante do programa continua.

A condição de continuidade é sempre calculada no início, isto significa que se for falsa desde a primeira vez o bloco de comandos nunca será executado.

O próximo exemplo faz uma tabela das raízes quadradas dos inteiros de um até 10. A variável *i* inicialmente vale um. Quando a expressão `i<=10` é calculada pela primeira vez, ela é verdadeira, logo os dois comandos seguintes são executados. Um desses comandos incrementa *i* em um. Logo a cada repetição o valor de *i* é aumentado. Quando ele se torna maior que dez, a condição `i<=10` passa a ser falsa e programa passa o comando `print` e termina.

```
1           #programa raizesquadradas.py
2           # -*- coding: cp1252 -*-
3           from __future__ import print_function
4           import math
5
6           print("\n\n\nTabela de Raízes Quadradas")
7           i = 1
8           while i<=10:
9               print('raiz de {0:2} = {1:.4}'.format(i,
            math.sqrt(i)))
10              i += 1
11          print('\n\n\n')
```

Exercícios.

30. Faça um programa que imprima na tela apenas os números impares entre 1 e 50.

31. Faça um programa que peça dois números inteiros e gere os números inteiros que estão no intervalo compreendido entre eles.

32. A sequência de Fibonacci é formada pelos números 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... (o próximo número é a soma dos anteriores). Faça um programa capaz de gerar a sequência até o valor seja maior que 500.

33. Faça um programa que peça uma nota, entre zero e dez. Mostre uma mensagem caso o valor seja inválido e continue pedindo até que o usuário informe um valor válido.

34. Faça um programa que imprima na tela os números de 1 a 20, um abaixo do outro.

35. Faça um programa que leia um nome de usuário e a sua senha e não aceite a senha igual ao nome do usuário, mostrando uma mensagem de erro e voltando a pedir as informações.

36. Faça um programa que leia e valide as seguintes informações:

    a. Nome: maior que 3 caracteres;
    b. Idade: entre 0 e 150;
    c. Salário: maior que zero;
    d. Sexo: 'f' ou 'm';
    e. Estado Civil: 's', 'c', 'v', 'd';

37. Supondo que a população de um país A seja da ordem de 80000 habitantes com uma taxa anual de crescimento de 3% e que a população de B seja 200000 habitantes com uma taxa de crescimento de 1.5%. Faça um programa que calcule e escreva o número de anos necessários para que a população do país A ultrapasse ou iguale a população do país B, mantidas as taxas de crescimento.

38. Altere o programa anterior permitindo ao usuário informar as populações e as taxas de crescimento iniciais. Valide a entrada e permita repetir a operação.

39. Um funcionário foi contratado no ano 2000. Em 2001 ele recebeu um aumento de 1,5% sobre o seu salário. A cada ano o aumento de salário foi sempre o dobro do ano anterior. Faça um programa que calcule o salário atual do funcionário. O programa deve pedir o ano, e calcular o salário naquele ano.

40. Faça um programa que peça uma quantidade indeterminada de números entre 0 e 100, e conte quantos deles estão nos intervalos: [0,25], [26,50], [51,75] e [76,100]. A entrada de dados deve terminar quando o número digitado estiver fora da faixa de 0 a 100.

41. Suponha que o cardápio de uma lanchonete seja:

Especificação	Código	Preço
Cachorro Quente	100	R$ 2,50
Bauru Simples	101	R$ 2,70
Bauru com ovo	102	R$ 3,00
Hamburger	103	R$ 3,50
Cheeseburger	104	R$ 3,60
Refrigerante	105	R$ 2,50

Faça um programa que leia o código dos itens pedidos e as quantidades desejadas. Calcule e mostre o valor a ser pago por item (preço*quantidade). O pedido será encerrado quando o código for zero. Com o encerramento do pedido o programa mostra o total geral.

Precisão da máquina: O número épsilon

A forma de representação de números reais em um computador possui limitações quanto ao tamanho máximo do número a ser armazenado, e em relação ao número de algarismos significativos que podem ser usados. Essas limitações dependem do tipo de máquina (calculadora, celular, notebook, etc.), do sistema operacional em uso e até da linguagem de programação que se está usando.

Para avaliar o número de algarismos significativos que um determinado sistema pode trabalhar, existe um número chamado "Épsilon da Máquina". O épsilon da máquina é um valor que mede o número de casas decimais da mantissa de um número real armazenado em binário.

Existe uma sequência de passos padronizada que leva a esse número que é descrito a seguir.

Primeiramente parte-se do número 1, que usando a representação de números reais em binário é *eps* = 0,1000 x 2^1. Em seguida o valor inicial é dividido por 2. No sistema decimal o resultado é 0,5 entretanto esse número é armazenado na memória da máquina como 0,1000 x 2^0. O ponto chave para se descobrir a quantidade de bits da mantissa é a expressão:

$$1 + eps$$

Para que essa operação seja efetuada no interior da máquina, é necessário que os dois números (o 1 e o *eps*) sejam escritos na mesma base para que, em seguida, a mantissa seja somada bit a bit. Assim o número de menor expoente é ajustado para ficar igual ao do maior expoente, assim a operação

$$0{,}1000 \times 2^1 + \mathbf{0{,}1000 \times 2^0}$$

se transforma em

$$0{,}1000 \times 2^1 + \mathbf{0{,}0100 \times 2^1} = 0{,}1100 \times 2^1$$

Observa-se que o bit do número de **menor** potência foi deslocado uma casa para a direita. Cada vez que a variável *eps* for dividida por 2, o bit será deslocado uma casa para a direita, até que não exista mais um bit disponível (no caso do exemplo acima são quatro bits disponíveis para a mantissa).

Quando não houverem bits disponíveis o número será arredondado para zero, assim

$$0{,}1000 \times 2^1 + \mathbf{0{,}0000 \times 2^1} = 0{,}1000 \times 2^1$$

e o resultado da operação será 1. Por definição o número Épsilon da Máquina é o menor número representável antes que se torne zero. Por isso, no programa `eps.py` abaixo que implementa o procedimento, o resultado é duplicado, pois as contas são feitas até que *eps* seja zero.

A informação mais importante a ser observada nesse número é a potência de 10, que indicará o número de algarismos significativos que o sistema consegue trabalhar. Por exemplo se *eps* = 2,27E-**12**, isso significa que o sistema pode trabalhar até com 12 algarismos significativos.

```
1           # programa eps.py
2
```

```
3           eps = 1.0
4
5           while 1+eps > 1:
6               eps /= 2
7
8           print 'eps =',2*eps
```

`While` aninhado

Usa-se o termo *aninhado* para se referir a blocos de instruções que estão dentro, isto é, recuados em relação a um outro bloco de instruções.

Dentre as instruções que estão dentro do bloco de instruções do `while`, qualquer instrução pode ser usada, inclusive uma outra instrução `while` com um novo bloco de instruções recuadas. Para ilustrar a ideia, o exemplo a seguir escreve a sequência de inteiros organizado em cinco linhas de seis colunas.

Primeiramente a variável chamada `contador`, usada para guardar o número que será escrito, é inicializada. Em seguida inicializa-se a *string* `s` que conterá o texto a ser mostrado na tela. Passa-se, então para as repetições, a variável `i` conta o número de linhas e será inicializada *apenas uma vez* durante toda a execução do programa.

Dentro do `while` a variável `j`,que conta o número de colunas, é inicializada. Assim a variável `j` será inicializada toda vez que o bloco de instruções do `while` for repetido. Em seguida vem um *novo* `while` (que é o `while` aninhado) que irá incrementar a variável `j`, e só terminará quando o `j` for maior que 6. Uma maneira de analisar o `while` aninhado é de se pensar "*para cada i, faz-se...*". Dessa forma tem-se que para cada valor de `i` (linha) faz-se `j` variar de 1 até 6. Para cada valor de `j` acrescenta-se o contador na *string* `s` (usando a função `str` para converter o tipo inteiro em *string*) e uma marca de tabulação (para que o número seguinte não fique colado no anterior), depois incrementa-se a variável que conta o número de colunas, e avariável que guarda o número inteiro.

Terminado o `while` interior, volta-se ao bloco de instruções do `while` mais externo. Nesse bloco acrescenta-se uma marca de fim de linha e incrementa-se o contador de linhas. Quando as cinco linhas tiverem sido criadas, isto é, a variável `i` for maior que cinco, a *string* `s` é escrita na tela.

```
1           # programa waninhado.py
2           #  -*- coding: utf-8 -*-
3           from __future__ import print_function
4
5           # inicializando contador e string
6           contador = 1
7           s = ''
8
9           i = 1
10          while i<=5: # while mais externo
11              j = 1
12              while j<=6: # while aninhado
13                  print(contador, end='\t')
```

```
14                  j += 1
15                  contador += 1
16              print(' ')
17              i += 1
```

Exercícios.

42. Faça um programa que, dado um conjunto de números quaisquer, determine o menor valor, o maior valor e a soma dos valores. O programa deve aceitar apenas números entre 0 e 100. Caso o número seja 0 o programa para de pedir os números e mostra o resultado. Caso o número esteja fora do intervalo de 0 a 100 ele deve dar uma mensagem de erro e pedir o número novamente, até que seja fornecido um número válido.

Repetição de comandos um número fixo de vezes com o `for`

Para se repetir um bloco de comandos um número fixo de vezes usa-se o comando `for`. Nesse comando defini-se uma variável, que pode ser usada como um contador, que vai ser diferente a cada repetição do bloco de comandos. Cabe ao programador definir o nome e a lista de valores possíveis para essa variável. A forma geral do comando `for` é a que segue:

`for` *variável* `in` *sequência*`:`
    comando 1
    comando 2
    etc.
    comando n
próximo comando

A sequência colocada na instrução `for` pode ser uma tupla, uma lista ou uma função que retorna uma tupla ou lista, como é o caso da função `range`.

O exemplo a seguir escreve o quadrado de todos os inteiros de 1 a 10. O comando `print` da última linha é executado dez vezes pois a função `range` gera uma lista de inteiros de um até dez. Assim, na primeira vez *i* vale um, na segunda vez *i* vale 2, e assim por diante até que *i* vale 10 e o programa termina. A variável *i* não precisa ser inicializada antes de ser usada no `for` pois o próprio comando `for` inicializa a variável.

```
1           # program quadrados.py
2           # -*- coding: cp1252 -*-
3
4           print('\n\nTabela de Quadrados\n')
5           for i in range(1,11):
6               print('{0:2} ao quadrado é {1:3}'.format(i,
            i*i))
7           print('\n')
```

Exercícios.

43. Faça um programa que imprima na tela apenas os números ímpares entre 1 e 50.

44. Faça um programa que peça dois números inteiros e gere os números inteiros que estão no intervalo compreendido entre eles.

45. A sequência de Fibonacci é formada pelos números 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... (o próximo número é a soma dos anteriores). Faça um programa capaz de gerar a sequência até o n-ésimo termo.

46. Faça um gerador de tabuada capaz de gerar a tabuada de qualquer número inteiro de 1 a 10. O usuário deve informar de qual número ele deseja ver a tabuada. A saída deve ser conforme o exemplo abaixo:

```
Tabuada de 5:
 5 X  1 =  5
 5 X  2 = 10
...
 5 X 10 = 50
```

47. Altere o Gerador de Tabuada que não necessariamente inicie de 1 e termine em 10, o valor inicial e final devem ser informados pelo usuário, veja o exemplo abaixo

```
Tabuada de -> 5
Começar por -> 7
Terminar em -> 4

Tabuada de 5
 5 X  4 = 20
 5 X  5 = 25
 5 X  6 = 30
 5 X  7 = 35
```

Deve-se verificar (e corrigir) o problema caso o usuário digite o valor inicial maior que o valor final.

48. Faça um programa que calcule o valor total de uma coleção de CD's. O programa deverá solicitar a quantidade de CD's, em seguida o valor de cada um.

49. Faça um programa que calcule e mostre a média aritmética de um número dado de notas. O programa deverá solicitar a quantidade de notas, depois solicitar cada nota para, em seguida, mostrar a média. (Dica. Armazene a soma das notas à medida que elas forem sendo digitadas).

50. Faça um programa que peça para *n* pessoas a sua idade, ao final o programa deverá mostrar a média das idades e classificar a turma como “jovem”, “adulta” ou “idosa” de acordo com a seguinte faixa etária: menor que 25 (jovem), entre 25 e 61 (adulta), e maior ou igual a 61 (idosa).

51. Faça um programa que peça 10 números inteiros, calcule e mostre a quantidade de números pares e a quantidade de números impares.

52. Faça um programa que simule uma eleição. Suponha que existem três candidatos (numerados 1, 2 e 3). O programa deverá perguntar o número de eleitores, em seguida pedir o voto de cada eleitor e por fim mostrar a quantidade de votos para cada candidato e

a quantidade de votos nulos.

53. Faça um programa que receba os seguintes dados de cinco cidades:

a. O nome da cidade;
b. O número de veículos; e
c. O número de acidentes de trânsito com vítimas

Ao final o programa deverá mostrar:

i. Qual a cidade com menor número de acidentes, e seu número de acidentes;
ii. Qual a cidade com maior número de acidentes, e seu número de acidentes;
iii. Qual a média de veículos nas cinco cidades juntas; e
iv. Qual a média de acidentes de trânsito com vítimas nas cidades com menos de 500 veículos.

Somatórios e Produtórios

Somatório simples

```
1           # programa somatorio1.py
2           # -*- coding: utf-8 -*-
3           from __future__ import print_function
4
5           S = 0
6           for i in range(1,11):
7               S += 1.0/i
8
9           print('O somatório é:{0} \n'.format(S))
```

Exercícios.

54. Faça um programa que peça dois números inteiros e gere os números inteiros que estão no intervalo compreendido entre eles, e no final mostre a soma dos números.

55. Sendo $S = 1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n$, faça um programa que calcule o valor de S com n termos.

56. Faça um programa que mostre o resultado de cada um dos termos da série abaixo, e no final mostre o somatório.

$$S = 1/1 + 2/3 + 3/5 + 4/7 + \ldots + n/m$$

Produtório simples

```
1           # programa produtorio1.py
2           # -*- coding: utf-8 -*-
3           from __future__ import print_function
4
5           P = 1.0  # nunca inicialize em zero
```

```
6           for i in range(1,11):
7               P *= 1.0/i
8
9           print('O produtório é:{0}\n'.format(P))
```

Exercícios.

57. Faça um programa que calcule o fatorial de um número inteiro fornecido pelo usuário. Ex. 4! = 4.3.2.1 = 24.

58. Faça um programa que peça dois números, base (número real) e expoente (número inteiro), calcule e mostre o primeiro número elevado ao segundo número. **Não utilize a função ou o operador de potenciação.**

Somatório com termo geral diferente

```
1           # programa somatorio2.py
2           # -*- coding: utf-8 -*-
3
4           S = 0
5           for i in range(1,11):
6               S += 1.0/i**2
7
8           print 'O somatório é:',S
```

Com menor esforço computacional

```
1           # programa somatorio3.py
2           # -*- coding: utf-8 -*-
3
4           S = 0
5           for i in range(1,11):
6               S += 1.0/(i*i)
7
8           print 'O somatório é:',S
```

Série de potências simples

```
1           # programa somatorio3_1.py
2           # -*- coding: utf-8 -*-
3
4           x = 1.0/2
5
6           S = 0
7           for n in range(0,11):
8               S += x**n
9
10          print 'O somatório é:',S
```

aproveitando o termo anterior

```
1           # programa somatorio4.py
2           # -*- coding: utf-8 -*-
3
4           x = 1.0/2
5
6           T = 1.0         # O primeiro termo x**0
7           S = T           # Inicializar como o primeiro termo
8           for n in range(1,11): # começa do segundo termo
9               T = x*T     # Obtem o novo tempo a partir do
            anterior
10              S += T
11
12          print 'O somatório é:',S
```

com o fatorial

```
1           # programa somatorio5.py
2           # -*- coding: utf-8 -*-
3
4           x = 1.0
5
6           T = 1.0                 # O primeiro termo x**0/0!
7           S = T                   # Inicializar com o primeiro
            termo
8           for n in range(1,11): # começa do segundo termo
9               T = x/n*T  # Obtém o novo termo a partir do
            anterior
10              S += T
11
12          print 'O somatório é:',S
```

exemplo do cosseno

```
1           # programa somatorio6.py
2           # -*- coding: utf-8 -*-
3
4           # Calcula o cos(x) por série de potência
5
6           x = 1.0
7
8           T = 1.0                 # O primeiro termo x**0/0!
9           S = T                   # Inicializar com o primeiro
            termo
10          for n in range(2,2*11,2): # começa do segundo
            termo
11              T = -T*x*x/(n*(n-1))  # Obtém o novo termo a
            partir do
                 anterior
12              S += T
```

```
13
14          print 'O somatório é:',S
```

*** For Aninhado ***

Exercícios

59. Faça um programa que mostre todos os primos entre 1 e 1000.

60. Altere o programa do execício anterior de tal forma que conte o número de operações de divisão necessárias para encontrar cada número primo, e o número total de divisões para descobrir todos os números primos.

Interrompendo as repetições do `while` e do `for`

Tanto o `while` como o `for` executam blocos de comandos repetidamente. Em ambos os casos o teste para decidir se o bloco de comandos será executado ocorre antes do início de cada repetição. Entretanto podem haver situações em que se deseja interromper o bloco de comandos no meio, antes de terminar de executar todos os comandos. Existem dois comandos para realizar a interrupção das repetições: o `break` e o `continue`.

Quando o comando `break` é executado no interior do bloco de comandos de um `while` ou de um `for` as repetições são interrompidas e a execução passa para o próximo comando depois desse `while` ou `for`.

Diferentemente, quando o comando `continue` é executado no interior do bloco de comandos de um `while` ou `for` a execução volta para o início desse `while` ou `for`.

Apesar de poder aparecer em qualquer parte de um programa como um comando qualquer, esses comandos só fazem sentido se estiverem dentro de um comando `if`.

O exemplo a seguir ilustra o uso do `break`. O teste no início do `while` sempre será verdadeiro o que faria com que o bloco de comandos fosse executado executado infinitamente. Entretanto há um teste para encerrar as repetições no final do bloco. Quando *y* for igual a três o `while` é interrompido e o valor de *y* é escrito na janela do terminal.

```
1           # programa interrompe.py
2           y=0
3           while True:
4               y = y+1
5               if y==3: break
6           print('y = {0}'.format(y))
```

Exercício.

61. Faça um programa que calcule o fatorial de um número inteiro fornecido pelo usuário. O programa deverá permitir ao usuário entrar com quantos números desejar. O usuário deverá indicar que não quer mais entrar com números, entrando com um número maior que 20.

62. Faça um programa que, dado um conjunto de números quaisquer, determine o menor valor, o maior valor e a soma dos valores. O programa deverá encerrar quando o usuário digitar o número 0. (dica: à medida que o usuário for entrando com os números armazene o maior valor, o menor valor e a soma parcial).

63. Faça um programa que peça um número inteiro e determine se ele é ou não um número primo (divisível apenas por 1 e por ele mesmo).

64. Faça um programa que liste todos os divisores do número (exceto 1 e ele mesmo), e, caso não tenha divisores, informe que é um número primo.

65. Faça um programa que calcule o número médio de alunos por turma. O programa deverá pedir a quantidade de turmas e, em seguida, pedir o número de alunos de cada turma. Caso a quantidade de alunos na turma seja negativa ou maior que 40, o programa deverá informar uma mensagem de erro e solicitar novamente o número de alunos na turma até que o usuário entre com um valor válido.

66. Faça um programa que simule uma compra de supermercado. O programa deverá receber um número arbitrário de valores referentes aos preços de mercadorias. Se o valor zero for informado, indica que a compra está encerrada. O programa solicita o valor pago em dinheiro e calcula o troco. A saída deve ser como o exemplo abaixo:

```
**** Nova Compra ******
Preço 1 -> 3.2
Preço 2 -> 5.5
Preço 3 -> 7.9
Preço 4 -> 0
Total: R$ 16.60

Valor pago em dinheiro -> 20

Dinheiro R$ 20.00
Troco: R$ 3.40
```

Após essa operação o programa deverá voltar ao ponto inicial para registrar a próxima compra. O programa terminará se um preço for negativo. (Dica: A função `exit()` encerra o programa)

O comando `for` com cláusula `else`.

Com a presença de uma instrução `break` no interior do bloco de comandos do `for`, esse bloco pode ser encerrado de duas formas: ou o bloco de comandos é repetido até o final, ou as repetições são interrompidas pela execução do `break`.

Em muitas situações, o que será feito depois dependerá de como o `for` foi encerrado. Para incluir a possibilidade fazer distinção entre esses dois casos existe a cláusula `else` para o `for`. O bloco de comandos no interior da clausula `else` só será executada se o

`for` não encerrar com `break`. Assim, o uso do `for` com `else` é:

```
for variável in seguência:
 Bloco de comandos do for
   if condição:
      Bloco de comandos a serem executados
      caso o for encerrar com esse break
      break
else:
   Bloco de comandos a serem executados se
   o for terminar normalmente
```

O programa `forelse.py` listado abaixo ilustra o uso dessa construção. Ele verifica, se um determinado número dado pelo usuário é menor que algum número de uma sequência previamente criada. Caso seja, ele diz qual o número da sequência que é maior que o número dado e encerra o `for` (com um `break`). Caso não seja, isto é, o `for` varre toda a sequência em busca de um número maior e não acha, ele informa esse fato ao usuário (na cláusula `else`).

```
1           # programa forelse.py
2           # coding: utf-8
3
4           from __future__ import print_function, division
5
6           L = (2,4,8,16,23,64,128,256,521,1024)
7
8           n = int(input('\nEntre com um número inteiro -> ')
            )
9
10          for i in L:
11            if i > n:
12              print('\nO número {0} é maior que
            {1}'.format(i,n))
13              break
14          else:
15            print('\n{0} é maior que todos os
            números'.format(n))
16
17          print('\nTerminando...\n')
```

Exercícios

67. Altere o programa `forelse.py` acima que inclua o número dado pelo usuário na lista, caso o número seja maior que todos os números da sequência.

68. Faça um programa que peça um número inteiro e determine se ele é ou não um número primo (Obs. Use o `for` com cláusula `else`).

Tratamento de Exceção com `try`, `except`, `else` e `pass`.

Nem todas os comandos que se deseja que um computador execute são possíveis de serem executados sempre. Um comando para gravar alguma coisa na memória pode não ser executado porque não tem memória suficiente disponível, ou a unidade de memória acabou de ser removida e não se tem mais acesso a ela. Em operações matemáticas, onde são usadas varias operações aritméticas tem-se o problema da divisão por zero e das funções cujos domínios são limitados.

O programa excecao.py na página 68 ilustra esse problema. É um programa que pede um número, e calcula o inverso da raiz quadrada dada por:

$$\frac{1}{\sqrt{x}} \tag{1}$$

em seguida imprime a mensagem "Finalizou bem".

Ao executar esse programa, se o usuário fornece um número negativo, ou zero conforme o exemplo abaixo, uma mensagem de erro aparece e o programa é interrompido. Diz-se que ocorreu uma *exceção*. Uma vez que o programa é interrompido, as linhas 7 em diante não são executadas.

```
>>> exec(open('excecao.py').read())
entre com x -> 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "excessao.py", line 6, in <module>
    y =  1/sqrt(x)
ZeroDivisionError: float division
```

```
1           # programa excecao.py
2           # coding: utf-8
3           from math import sqrt
4           x = float(input("entre com x -> "))
5
6           y =  1/sqrt(x)
7
8           print('\nA inversa da raiz quadrada de {0} é
            {1:.5f}\n'.format(x,y))
9
10          print('Finalizou bem\n\n')
```

Entretanto, em muitas situações deseja-se que o programa apenas "passe por cima" da linha problemática e continue executando. A esse processo de dar um tratamento diferente para uma linha, ou linhas, que podem potencialmente ter problemas, chama-se de *tratamento de exceção*.

O comando que marca uma linha ou bloco de linhas para o tratamento de exceção é o comando `try` com uma ou mais cláusulas `except`. Esse comando pode vir acompanhado de uma clausula opcional `else`. A estrutura é a que segue:

```
try:
```

bloco de comandos que podem gerar uma exceção (dar problema)
`except:`
bloco de comandos a serem executados se houver uma exceção
`else:`
bloco de comandos a serem executados se não houver uma exceção
próxima linha depois do try (sempre será executada)

As linhas que estão entre o `try` e o `except` serão executadas. Quando a primeira delas der algum problema, isto é, gerar uma exceção, o bloco de comandos que vem depois do `except` será executado. Caso não haja nenhum problema, isto é, nenhuma exceção for gerada o bloco de comandos depois do `else` será executado. Ocorrendo, ou não, uma exceção, a próxima linha depois do `try` em diante serão executadas e o programa fluirá normalmente.

O programa `excecao1.py` a seguir ilustra o uso dessa estrutura para evitar o problema anterior. O cálculo da fórmula, na linha 7, está dentro de um comando `try`. Caso ocorra uma exceção, o comando `pass`, na linha 9 será executado. O comando `pass` não faz nada, apenas passa a execução para a primeira linha depois do `try`.

Caso o número dado pelo usuário esteja bom, então o `y` é calculado e o comando da cláusula `else` será executado imprimindo na tela o resultado das operações aritméticas. Independente de ocorrer uma exceção ou não, a frase "Finalizou bem" vai aparecer na tela. Caso não houvesse o tratamento de exceção, toda vez que uma exceção ocorresse a linha "Finalizou bem" não seria escrita.

```
1           # programa excecao1.py
2           # coding: utf-8
3           from math import sqrt
4           x = float(input("entre com x -> "))
5
6           try:
7               y =  1/sqrt(x)
8           except:
9               pass
10          else:
11              print('\nA inversa da raiz quadrada de {0} é
            {1:.5f}\n'.format(x,y))
12
13          print('Finalizou bem\n\n')
```

O programa `excecao1.py` não informa que ocorreu um problema, simplesmente continua a execução como se nada tivesse acontecido. Seria importante que o programa avisasse que o problema ocorreu. O programa `excecao2.py` faz isso colocando na linha 9 o comando `print` que informa que ocorreu um problema na avaliação da função.

```
1           # programa excecao2.py
2           # coding: utf-8
3           from math import sqrt
4           x = float(input("entre com x -> "))
```

```
5
6           try:
7               y =  1/sqrt(x)
8           except:
9               print('\nOcorreu um erro no cálculo da inversa
            da raiz quadrada')
10          else:
11              print('\nA inversa da raiz quadrada de {0} é
            {1.5f}\n'.format(x,y)
12
13          print('Finalizou bem\n\n')
```

No cálculo da equação 1 da página 68 existem dois problemas diferentes que podem ocorrer: a) pode ocorrer uma exceção de “divisão por zero”, se o denominador da fração for zero; b) pode ocorrer uma exceção de “valor fora de domínio” para a raiz quadrada, se o número for negativo.

Para que o tratamento de exceção seja diferente em cada um dos tipos de erro possíveis, colocam-se duas cláusulas `except`, uma para cada tipo de exceção. Para separar qual bloco irá tratar qual exceção, coloca-se na cláusula `except` um parâmetro que indica o tipo de exceção. Para divisão por zero o parâmetro é `ZeroDivisionError`, e para valor fora de domínio da função é `ValueError`.

O programa excecao3.py ilustra o tratamento separado dois tipos de exceção. Caso a exceção seja uma divisão por zero, o que vem depois da linha 8 até o próximo `except` será executado. Caso seja um erro no domínio da função `sqrt` da linha 11 até o próximo `except` será executado. Se ocorrer algum outro tipo de exceção não prevista, da linha 14 até o `else` será executado.

```
1           # programa excecao3.py
2           # coding: utf-8
3           from math import sqrt
4           x = float(input("entre com x -> "))
5
6           try:
7               y =  1/sqrt(x)
8           except ZeroDivisionError:
9               print('\nO valor de x não pode ser zero por
            causa da inversa.')
10          except ValueError:
11              print('\nO valor de x não pode ser negativo
            por causa da raiz quadrada.')
12          except:
13              print('\nOcorreu um erro desconhecido no
            cálculo da inversa da raiz quadrada')
14          else:
15              print('\nA inversa da raiz quadrada de {0} é
            {1:.5f} \n'.format(x,y))
16
17          print('Finalizou bem\n\n')
```

Os programas anteriores possuem ainda uma fragilidade. Na linha 4, ao pedir um número, o usuário pode entrar com alguma informação não numérico, como, por exemplo um nome. O programa excecao4.py resolve esse problema colocando a linha dentro de um comando try. Caso o usuário digite alguma coisa inválida, o tratamento de exceção mostra uma mensagem e pede que usuário entre com o número novamente. O comando `try` está dentro de um **`while`** cuja condição é sempre verdadeira (isto é, um laço infinito). Só será possível sair do `while` a partir de um comando break. O comando break só será executado, e, consequentemente o restante do programa será executado, se o usuário digitar um número válido.

```
1           # programa excecao4.py
2           # coding: utf-8
3           from math import sqrt
4
5           while True:
6               try:
7                   x = float(input('entre com x -> '))
8               except:
9                   print('\n\nEntre com um número
            válido\nTente novamente\n')
10              else:
11                  break
12
13          try:
14              y =  1/sqrt(x)
15          except ZeroDivisionError:
16              print('\nO valor de x não pode ser zero por
            causa da inversa.')
17          except ValueError:
18              print('\nO valor de x não pode ser negativo
            por causa da raiz quadrada.')
19          except:
20              print('\nOcorreu um erro desconhecido no
            cálculo da inversa da raiz quadrada')
21          else:
22              print('\nA inversa da raiz quadrada de {0} é
            {1:.5f} \n'.format(x,y))
23
24          print('Finalizou bem\n\n')
```

Função Definição e Sub-Rotina

Além das funções matemáticas e da Função Lambda, o programador pode criar funções mais complexas chamadas de Função Definição. A diferença entre uma Função Declaração, ou Função Lambda e a Função Definição é que na primeira só se pode usar uma fórmula na criação da função, ao passo que na segunda, a Função Definição pode-se usar qualquer comando na sua definição. A forma mais simples de se criar uma Função Definição é:

```
def nome da função(variável 1, variável 2, ... , variável n):
    comandos
    return fórmula
```

O valor retornado pela função será o resultado da fórmula. A linha com o comando `return` é opcional e pode ser omitida, assim a função não retorna valor algum. Mutas vezes o que se deseja em uma função não é o seu valor final, mas o *efeito colateral* do seu uso, como, por exemplo uma mensagem escrita na tela ou um arquivo gravado em disco. Com frequência uma função que não retorna valor é chamada de *sub-rotina*.

Uma função, ou uma sub-rotina, pode ser, em si, um programa completo em que quaisquer comandos podem ser usados, e não apenas operações aritméticas. Além disso pode receber desde nenhum argumento, até uma quantidade desconhecida de argumentos. Esses recursos permitem que programas bastante complexos possam ser desmembrados em pedaços mais simples de entender e administrar.

O programa a seguir ilustra um exemplo de uma *sub-rotina* (função que não retorna valor algum). Essa sub-rotina não possui argumento de entrada, simplesmente escreve a frase "programando" dez vezes.

```
1           # escreve.py
2           # A sub-rotina escreve a frase 'programando...'
3           # na tela dez vezes
4
5           def escreve():
6               for i in range(10):
7                   print('programando...')
```

Os parêntesis são importantes. Se a função não recebe parâmetros os parêntesis ficam vazios.

Para retornar mais de um valor, uma função pode retornar uma tupla. Nesse caso quando a função for chamada, se colocar apenas uma variável no lado direito da expressão, a variável será uma tupla. Alternativamente pode-se colocar uma variável para cada elemento da tupla.

O exemplo a seguir cria uma função, chamada `gms`, que toma um ângulo em graus (e décimos de grau), e converte em graus, minutos e segundos.

```
8           # programa gms.py
9           # coding: utf-8
10          from __future__ import print_function, division
11
12          def gms(x):
13              graus = int(x)
14              x = x-graus
15              x = x*60
16              minutos = int(x)
17              x = x-minutos
18              segundos = x*60
```

```
19              return (graus, minutos, segundos)
20
21          numero = 3.62
22          print('\nO ângulo {0} fica:'.format(numero))
23          gr, minu, seg = gms(numero)
24          print('{0} graus {1} minutos e {2:.2f}
             segundos\n'.format(gr,minu,seg))
```

Quando a função é chamada ela retorna três números que ficam guardadas nas variáveis `gr`, `minu` e `seg`.

Exercícios

69. Faça um programa que cria, e usa, uma **Função Declaração (**`lambda`**)**. A função deve receber três argumentos e retornar a soma desses argumentos. O programa principal deverá pedir três números e escrever a soma usando a Função Declaração criada.

70. Faça um programa que cria, e usa, uma **Função Definição** (`def`). A função deve receber três argumentos e retornar a soma desses argumentos. O programa principal deverá pedir três números e escrever a soma usando a Função Declaração criada.

71. Faça um programa, com uma função `def`, que necessite um argumento. A função retorna o valor de caractere 'P', se seu argumento for positivo, e 'N' se seu argumento for zero ou negativo. O programa principal deve pedir o número e escrever o resultado da função.

72. Faça um programa com uma função (`def`) chamada `somaImposto`. A função possui dois parâmetros: *taxa*, que é a quantia de imposto sobre vendas expressa em porcentagem e *custo*, que é o custo de um item antes do imposto. A função "altera" o valor de custo para incluir o imposto sobre vendas. O programa principal deverá pedir o valor da taxa e do custo e escrever o resultado da função.

73. Faça uma função que converta da notação de 24 horas para a notação de 12 horas. Por exemplo, o programa deve converter 14:25 em 2:25 P.M. A entrada é dada em dois inteiros. O programa principal deve pedir os dois inteiros e escrever o resultado. (Dica: A formatação de *string* com o método `format` normalmente usada na funçaõ `print`, também pode ser usada no comando `return`)

74. Faça um programa que use a função valorPagamento para determinar o valor a ser pago por uma prestação de uma conta. O programa deverá solicitar ao usuário o valor da prestação e o número de dias em atraso e passar estes valores para a função valorPagamento, que calculará o valor a ser pago e devolverá este valor ao programa que a chamou. O cálculo do valor a ser pago é feito da seguinte forma. Para pagamentos sem atraso, cobrar o valor da prestação. Quando houver atraso, cobrar 3% de multa, mais 0,1% de juros por dia de atraso.

75. Faça uma função que informe a quantidade de dígitos de um determinado número inteiro informado.

76. Construa uma função que receba uma data no formato *DD/MM/AAAA* e devolva uma string no formato *DD de mês (por extenso) de AAAA*.

77. Altere o programa anterior de tal forma que dê a mensagem "Erro: data nula.

Observe a formatação pedida" e peça novamente para entrar com a data, até que o usuário entre com uma data válida.

78. Construa uma função chamada `iterador` que receba dois argumentos. O primeiro argumento é uma *função de iteração*, e o segundo um número real x_0. A função deverá retornar o valor da décima iteração da função de iteração. O programa principal chama a função e escreve o resultado na tela.

Observações sobre as variáveis de Funções Definição e de Sub-rotinas

1. Dentro de um único arquivo `.py` podem ser definidas várias funções.

2. Dentro da definição de uma Função Definição ou de uma sub-rotina podem ser definidas variáveis como se ela fosse um programa qualquer. As variáveis definidas dentro de uma função são variáveis *locais*, isto é, só existem dentro da função. Quando a função ou sub-rotina terminar de executar as variáveis são descartadas e o espaço de memória usada por elas é liberado. Assim, variáveis com o mesmo nome podem ser usadas em diferentes funções sem conflito. Se duas funções possuírem uma variável com o nome *x*, por exemplo, a variável *x* de cada função ocupará um espaço diferente na memória. O mesmo acontece no ambiente do terminal, caso exista uma variável com o nome *x*, por exemplo, uma variável com o nome *x* dentro de uma função não alterará o valor da variável do ambiente, e vice e versa.

3. Além das variáveis criadas dentro da função, os argumentos da função também são variáveis locais da função.

4. Quando uma função é chamada, os argumentos que são números, tuplas e strings são passadas para a função *por valor*, isso significa que apenas *cópias* do conteúdo das variáveis são passadas para a função. Na prática isto significa que as variáveis do programa principal que são colocadas como argumentos da função não podem ser alteradas pela função já que o que é passado para a função é apenas uma cópia do valor da variável. Entretanto listas e dicionários são passados *por referência*, isto é, as próprias listas são passadas para a função. Isso significa que caso uma lista seja alterada dentro de uma função, ela continuará alterada depois que a função terminou a execução.

O exemplo a seguir ilustra essa diferença, cria-se a função `funcao` que recebe três parâmetros `num`, `st` e `lis` e altera esses valores. Em seguida essa função é chamada com três variáveis `a`, `s` e `L`. Em seguida as variáveis são escritas na tela: os conteúdos de `a` e `s` não foram alteradas pela função, mas o conteúdo de `L` foi.

```
1           # passagem.py
2           # from __future__ import print_function
3
4           def funcao(num, st, lis):
5               num = 2
6               st = 'novo teste'
7               lis[0]=2
8
```

```
9           a = 1
10          s = 'teste'
11          L = [1]
12
13          funcao(a,s,L)
14
15          print(a) # a=1
16          print(s) # s='teste'
17          print(L) # L=[2]
```

5. Uma variável pode ser compartilhada entre vários programas, nesse caso ela é chamada de variável *global*. Pode-se definir uma variável como global usando-se a linha

`global` *nome da variável*

Essa linha deve ser colocada antes do uso da variável em cada função que compartilhar a variável.

O exemplo a seguir possui uma variável `gg` compartilhada pelas funções `duplicar` e `triplicar`, que duplicam e triplicam o valor de `gg` respectivamente. O programa principal inicializa `gg`, depois chama cada uma das funções e manda escrever na janela do terminal o valor da variável.

```
1           # programa global.py
2           # from __future__ import print_function
3
4           def duplicar():
5             global gg
6             gg *= 2
7
8           def triplicar():
9             global gg
10            gg *= 3
11
12          gg = 5
13
14          duplicar()
15          print(gg)   #escreve 10
16          triplicar()
17          print(gg)   #escreve 30
```

Cuidado. Deve-se evitar ao máximo o uso de variáveis globais pois elas podem causar erros difíceis de programação difíceis de serem descobertos.

Documentação de funções e sub-rotinas através de *doc strings*

No início da definição da função, logo após a primeira linha pode-se colocar, entre aspas triplas ou apóstrofes triplos, uma descrição da função. Essa descrição pode se estender por várias linhas e chama-se "*doc string*". Normalmente o texto colocado ali é uma descrição do que a função faz, e dá orientações de como usá-la. É uma boa prática colocar esses comentários na declaração de uma função, pois isso pode vir a ajudar a explicar como usá-la quando se passa muito tempo sem se trabalhar com ela. Para se ler a *doc string* sem precisar olhar o código fonte usa-se, no terminal, o comando

```
>>> help( nome da função )
```

O nome da função é sem os parêntesis.

Para exemplificar, o programa abaixo, salvo num arquivo chamado `mat.py`, define a função `sqr(x)`.

```
1           # programa mat.py
2           # -*- coding: utf-8 -*-
3           def sqr(x):
4               '''
5               A função \"sqr(x)\" eleva o
6               número x ao quadrado. O valor
7               retornado é do mesmo tipo do
8               valor dado.
9               '''
10              return x*x
```

Ao executar esse arquivo a partir do terminal a função é criada. Em seguida executa-se a função e se consulta a sua *doc string*.

```
>>> execfile('mat.py')
>>> sqr(4)
16
>>> help(sqr)

sqr(x)
    A função "sqr(x)" eleva o
    número x ao quadrado. O valor
    retornado é do mesmo tipo do
    valor dado.
```

Funções com coeficientes

Muitas funções possuem, além das variáveis, coeficientes que definem não apenas uma função, mas toda uma família de funções. Um exemplo típico é a equação da parábola dada por $p(x)=Ax^2+Bx+C$. Onde x é a variável e A, B e C são os coeficientes da

parábola. Normalmente os coeficientes são números fixos conhecidos que determinam a parábola.

Do ponto de vista computacional, para se criar uma função com coeficientes são necessárias duas etapas:

1) cria-se uma descrição da família, ou classe, de funções, onde se descreve a relação entre as variáveis e os coeficientes, e depois
2) cria-se a função propriamente dita, onde são atribuídos os valores de cada coeficiente.

A sintaxe usada para se criar uma função com coeficientes é:

```
class nome da família de funções:
    '''
    Uma doc string (opcional)
    '''
    def __init__(self,coeficiente 1, coeficiente 2, ... ):
        self.coeficiente 1 = coeficiente 1
        self.coeficiente 2 = coeficiente 2
          etc.
    def __call__(self,variável 1, variável 2, ...):
        Cálculos com as variáveis e os self.coeficientes.
        return resultado
```

Na primeira linha define-se o nome da classe, ou família de funções. A exemplo das funções normais, pode-se incluir uma *doc string* para orientar quem vai usar essa classe de funções. Em seguida existem duas funções definição: uma chamada `__init__` cujo objetivo é definir os nomes dos coeficientes, e outra chamada `__call__` que faz a relação entre as variáveis e os coeficientes e retorna o resultado.

É importante observar que:

1. Elas estão recuadas em relação à palavra `class`. Isso indica que essas estão dentro da definição da classe de funções, isto é, fazem parte da definição da classe de funções.
2. O *primeiro parâmetro* de cada uma é a palavra `self`. Essa é a palavra que faz a ligação das variáveis definidas dentro de uma função com as variáveis definidas dentro de outra. Por serem funções que fazem parte de uma classe, e possuem o parâmetro `self`, essas funções são chamadas de *métodos*.
3. Elas começam e terminam com dois caracteres sublinhados. Isso indica que são métodos (funções) internos da classe, eles não serão chamados diretamente quando a classe for usada.
4. Dentro da função `__call__` todos os nomes dos coeficientes tem que vir precedidos da palavra `self`.

No segundo passo, para efetivamente se criar a função, com os valores dos

coeficientes definidos, usa-se a sintaxe

nome da função = *nome da família de funções*(*valores iniciais dos parâmetros*)

O resultado desse comando de atribuição é uma função (assim como ocorre na definição de uma função lambda). Os valores entre parêntesis são os valores iniciais dos coeficientes, que podem ser alterados se necessário posteriormente.

Para se chamar a função usa-se a sintaxe usual para funções

nome da função(`variáveis`)

Para alterar o valor de um coeficiente, uma vez criada a função usa-se a sintaxe:

nome da função.*nome do coeficiente* = *novo valor*

Nesse caso não se usa a palavra `self`, mas sim apenas o nome do coeficiente.

Assim como no caso das funções, caso seja necessário ver a doc string de uma classe de funções já criada usa-se, no terminal, o comando print, com a sintaxe

`help(`*nome da família de funções*`)`

O exemplo a seguir cria uma classe de funções chamada `reta`. A equação geral da reta é $ax+b$ onde *a* e *b* são os coeficientes e *x* é a variável da equação da reta.

```
1           #programa: reta.py
2           # -*- coding: utf-8 -*-
3
4           class reta:
5               '''
6               Cria a classe de retas com equação a*x+b
7               a sintaxe é reta(a,b) onde a e b são os
8               coeficientes da reta.
9               '''
10              def __init__(self,a,b):
11                  self.a = a
12                  self.b = b
13              def __call__(self,x):
14                  return self.a*x+self.b
15
16          r1 = reta(1,1)
17          print(r1(0)) #resultado 1
18          print(r1(1)) #resultado 2
19          r1.b = 0     #modifica o coeficiente linear da
```

```
            reta
20          print(r1(0)) #resultado 0
21          help(reta)
```

Nas linhas de 4 a 14 define-se a classe chamada `reta`. A linha 16 cria a reta $r1(x)=x+1$. Em seguida (linhas 17 e 18) calculam-se os valores dessa reta nos pontos x=0 e x = 1.

A linha 19 altera o valor do coeficiente b, tornando-o zero. Depois calcula (linha 20) o valor da reta r1, com o coeficiente b=0 no ponto x=0.

A última linha manda imprimir a *doc string* na tela.

Cronometrando um Programa

Muitas vezes se deseja marcar quanto tempo um programa leva para executar, principalmente quando se está comparando métodos diferentes para resolver o mesmo tipo de problema. A maneira de fazer isso é de marcar o instante que o programa começou e o instante que o programa terminou. A diferença entre esses dois instantes é a duração do programa, ou trecho de programa. A função que marca o instante que o programa passa por um determinado ponto é a função

`datetime.now()`

que se encontra no módulo `datetime` (mesmo nome da função).

Essa função deve ser chamada no início do trecho de programa a ser cronometrado, e o resultado dessa função armazenado em uma variável. Ao final do trecho ela é chamada novamente para obter o instante que o trecho terminou. Em seguida calcula-se a diferença entre esses instantes. De forma geral a estrutura para se cronometrar o tempo de execução de um programa é:

comandos e linhas iniciais

```
import datetime
```
comandos não cronometrados

```
inicio = datetime.datetime.now()
```
comandos a serem cronometrados

```
fim = datetime.datetime.now()
tempoTranscorrido = fim-inicio
print(tempoTranscorrido)
```

No nome da função é necessário escrever *datetime* duas vezes porque o primeiro é o nome do módulo (da importação indireta) e o segundo é parte do nome da função.

Caso se necessite separar as parcelas do tempo, a variável `tempoTranscorrido` possui três propriedades úteis: `days`, que é o número de dias, `seconds`, que é o número de segundos transcorridos e `microsseconds`, que é o número de microssegundos.

Raízes de Funções

*** parei aqui ***

O Método do Ponto Fixo

O programa mínimo.

```
1           # programa: pontoFixo1.py
2           # --*-- coding: utf-8  --*--
3
4           # o Método do Ponto Fixo
5
6           tolerancia = 1.0e-5
7
8           # f(x)=x**2 + x - 6
9           phi = lambda x: (6-x)/x
10
11          x = 1.0
12          while True:
13              xNovo = phi(x)
14              erro = abs(xNovo-x)
15              x = xNovo
16              if erro < tolerancia: break
17
18          print 'Uma raiz é',x
19          print 'Com erro menor que',erro
```

Método do ponto fixo com contador.

```
1           # programa: pontoFixo2.py
2           # --*-- coding: utf-8  --*--
3
4           # o Método do Ponto Fixo
5
6           tolerancia = 1.0e-5
7           numeroMaximoDeIteracoes = 100
8
9           # f(x)=x**2 + x - 6
10          phi = lambda x: (6-x)/x
11
12          x = 1.0
13
14          contador = 0
15          while True:
16              xNovo = phi(x)
17              erro = abs(xNovo-x)
18              contador += 1
```

```
19              x = xNovo
20              if erro<tolerancia: break
21              if contador >= numeroMaximoDeIteracoes:
22                  print 'O método não convergiu

                 em',contador,'iterações'
23                  quit()
24
25          print 'Uma raiz é',x
26          print 'em', contador, 'iterações'
27          print 'com erro menor que',erro
```

Com estimativa da derivada de phi.

```
1           # programa: pontoFixo3.py
2           # --*-- coding: utf-8  --*--
3
4           # o Método do Ponto Fixo
5
6           tolerancia = 1.0e-5
7           numeroMaximoDeIteracoes = 100
8
9           # f(x)=x**2 + x - 6
10          phi = lambda x: (6-x)/x
11
12          x0 = 1.0
13
14          x = phi(x0)
15          contador = 1
16          while True:
17              xNovo = phi(x)
18              derivadaDePhi = (xNovo-x)/(x-x0)
19              erro = abs(xNovo-x)
20              contador += 1
21              x0 = x
22              x = xNovo
23              if erro<tolerancia: break
24              if contador >= numeroMaximoDeIteracoes:
25                  print 'O método não convergiu
            em',contador,'iterações'
26                  quit()
27
28          print 'Uma raiz é',x
29          print 'em', contador, 'iterações'
30          if derivadaDePhi>0.5:
31              print 'Estimativa de erro ruim'
32          else:
33              print 'com erro menor que',erro
```

Método de Newton-Raphson.

```
1           # programa: pontoFixo4.py
2           # --*-- coding: utf-8  --*--
3
4           # o Método de Newton-Raphson
5
6           tolerancia = 1.0e-5
7           numeroMaximoDeIteracoes = 10
8
9           f = lambda x: x*x + x - 6
10          flinha = lambda x: 2*x+1
11
12          x = 1.0
13
14          contador = 0
15          while True:
16              delta = f(x)/flinha(x)
17              xNovo = x-delta
18              contador += 1
19              x = xNovo
20              if abs(delta)<tolerancia: break
21              if contador >= numeroMaximoDeIteracoes:
22                  print 'O método não convergiu
            em',contador,'iterações'
23                  quit()
24
25          print 'Uma raiz é',x
26          print 'em', contador, 'iterações'
27          print 'com erro menor que',abs(delta)
```

Homotopia.

```
1           # programa: pontoFixo5.py
2           # --*-- coding: utf-8  --*--
3
4           # o Método de Newton-Raphson com Homotopia
5
6           tolerancia = 1.0e-5
7           numeroMaximoDeIteracoes = 10
8           nc = 10 # Número de incrementos de c
9
10          # função que se deseja achar uma raiz
11          f = lambda x: x*x + x - 6
12          flinha = lambda x: 2*x+1
13
14
15          # função auxiliar de raiz conhecida
16          h = lambda x: x-1
17          hlinha = lambda x: 1
18          x = 1.0
```

```
19
20          for i in range(1,nc+1):
21              c = float(i)/nc
22              contador = 0
23              while True:
24                  delta = ((1-c)*h(x)+c*f(x))/((1-c)
            *hlinha(x)+c*flinha(x))
25                  xNovo = x-delta
26                  contador += 1
27                  x = xNovo
28                  if abs(delta)<tolerancia: break
29                  if contador >= numeroMaximoDeIteracoes:
30                      print 'O método não convergiu
            em',contador,'iterações'
31                      quit()
32
33              print 'Para c =',c,' a raiz é',x
34              print 'em', contador, 'iterações'
35              print 'com erro menor que',abs(delta),'\n'
36
37          print 'Uma raiz é',x
```

O Método da Bisseção

O programa mínimo.

```
1           # programa bisseccao1.py
2           # --*-- coding: utf-8 --*--
3
4           tol = 1.0e-5
5
6           f = lambda x: x*x+x-6
7
8           a = 0.0
9           b = 3.5
10          while abs(b-a)/2.0>tol:
11              m = (a+b)/2.0
12              if f(a)*f(m) < 0:
13                  b = m
14              else:
15                  a = m
16
17          print 'a raiz é ', m
18          print 'com erro menor que', abs(b-a)/2.0
```

Incluindo o fato de m cair exatamente na raiz.

```
1           # programa bisseccao1_5.py
```

```
2           # --*-- coding: utf-8 --*--
3
4           tol = 1.0e-5
5
6           f = lambda x: x*x+x-6
7
8           a = 0.0
9           b = 3.5
10
11          erro = abs(b-a)/2.0
12          while erro>tol:
13              m = (a+b)/2.0
14              erro /= 2.0
15              if f(a)*f(m) < 0:
16                  b = m
17              elif f(a)*f(m) >0:
18                  a = m
19              else:
20                  erro = tol
21                  break
22
23
24          print 'a raiz é ', m
25          print 'com erro menor que', erro
```

O programa contando o número de iterações.

```
1           # programa bisseccao2.py
2           # --*-- coding: utf-8 --*--
3
4           tol = 1.0e-5
5
6           f = lambda x: x*x+x-6
7
8           a = 0.0
9           b = 3.5
10          contador = 0
11          erro = abs(b-a)/2.0
12          while abs(b-a)/2.0>tol:
13              contador += 1
14              m = (a+b)/2.0
15              erro /= 2.0
16              if f(a)*f(m) < 0:
17                  b = m
18              elif f(a)*f(m) > 0:
19                  a = m
20              else:
21                  erro = tol
22                  break
23
24          print 'a raiz é ', m
```

```
25          print 'com erro menor que', erro
26          print 'em',contador,' iterações'
```

Cronometrando o programa.

```
1           # programa bisseccao3.py
2           # --*-- coding: utf-8 --*--
3
4           import datetime
5
6           tol = 1.0e-5
7
8           f = lambda x: x*x+x-6
9
10          a = 0.0
11          b = 3.5
12
13          inicio = datetime.datetime.now()
14
15          contador = 0
16          erro =  abs(b-a)/2.0
17          while: erro > tol
18              contador += 1
19              m = (a+b)/2.0
20              erro /= 2.0
21              if f(a)*f(m) < 0:
22                  b = m
23              elif f(a)*f(m) > 0:
24                  a = m
25              else:
26                  erro = tol
27                  break
28
29          fim = datetime.datetime.now()
30          print 'a raiz é ', m
31          print 'com erro menor que', erro
32          print 'em',contador,' iterações'
33          print 'com tempo de',fim-inicio
```

Implementação com usando o `for`.

```
1           # programa bisseccao4.py
2           # -*- coding: utf-8 -*-
3
4           import datetime
5           import math
6
7           tol = 1.0e-5
```

```
8           interrompido = False
9
10          f = lambda x: x*x+x-6
11
12          a = 0.0
13          b = 3.5
14
15          inicio=datetime.datetime.now()
16
17          n = int(math.ceil((math.log(b-a)-math.log(tol)
            )/math.log(2
                 )-1))
18          for contador in range(n):
19             m = (a+b)/2.0
20             if f(a)*f(m)<0:
21                 b = m
22             elif f(a)*f(m)>0:
23                 a = m
24             else:
25                 interrompido = True
26                 break
27          if interrompido: erro = tol
28          else: erro = (b-a)/2.0
29
30          fim=datetime.datetime.now()
31          print 'A raiz é',m
32          print 'Com erro de', erro
33          print 'Em',n,'iterações'
34          print 'com tempo de', fim-inicio
```

Diminuindo o esforço computacional.

```
1           # programa bisseccao5.py
2           # --*-- coding: utf-8 --*--
3
4           tol = 1.0e-5
5
6           f = lambda x: x*x+x-6
7
8           a = 0.0
9           b = 3.5
10
11          fa = f(a)
12          contador = 0
13          erro = abs(b-a)/2.0
14          while erro>tol:
15              contador += 1
16              m = (a+b)/2.0
17              erro /=2.0
18              fm = f(m)
19              if fa*fm < 0:
```

```
20                  b = m
21              elif fa*fm > 0:
22                  a = m
23                  fa = fm
24              else:
25                  erro = tol
26                  break
27
28          print 'a raiz é', m
29          print 'com erro menor que', erro
30          print 'em',contador,'iterações'
```

O algoritmo Zbrac.

```
1           # programa zbrac.py
2           # --*-- coding: utf-8 --*--
3
4           maxiter = 50
5
6           f = lambda x: x*x+x-6
7           a = 0.0
8           b = 0.1
9
10          contador = 0
11          fa = f(a)
12          fb = f(b)
13          while fa*fb>0:
14              contador += 1
15              if contador>maxiter:
16                  print '\"zbrac\" não conceguiu cercar uma
            raiz                                    em',maxiter,'
            tentativas'
17                  quit()
18              if abs(f(a))<abs(f(b)):
19                  a -= b-a
20                  fa = f(a)
21              else:
22                  b += b-a
23                  fb = f(b)
24
25          print 'a =',a,' b =',b,' em',contador,'
            tentativas'
```

Bisseção com Zbrac.

```
1           # programa bisseccao6.py
2           # --*-- coding: utf-8 --*--
3
4           maxiter = 10
```

```
5           tol = 1.0e-5
6
7           f = lambda x: x*x+x-6
8           a = 0.0
9           b = 0.1
10
11          contador = 0
12          fa = f(a)
13          fb = f(b)
14          while fa*fb>0:
15              contador += 1
16              if contador>maxiter:
17                  print '\"zbrac\" não conceguiu cercar uma
            raiz em',maxiter,' tentativas'
18                  quit()
19              if abs(f(a))<abs(f(b)):
20                  a -= b-a
21                  fa = f(a)
22              else:
23                  b += b-a
24                  fb = f(b)
25
26          erro = abs(b-a)/2.0
27          while erro>tol:
28              contador += 1
29              m = (a+b)/2.0
30              erro /= 2.0
31              fm = f(m)
32              if fa*fm < 0:
33                  b = m
34              elif fa*fm>0:
35                  a = m
36                  fa = fm
37              else:
38                  erro = tol
39                  break
40
41          print 'a raiz é', m
42          print 'com erro menor que', erro
43          print 'em',contador,'iterações'
```

Melhorando o problema do UNDERFLOW.

```
1           # programa bisseccao7.py
2           # --*-- coding: utf-8 --*--
3
4           def sinal(x):
5               if x>0: return 1
6               elif x<0: return -1
```

```
7               else: return 0
8
9           maxiter = 10
10          tol = 1.0e-5
11
12          f = lambda x: x*x+x-6
13          a = 0.0
14          b = 0.1
15
16          contador = 0
17          fa = f(a)
18          fb = f(b)
19          while fa*fb>0:
20              contador += 1
21              if contador>maxiter:
22                  print '\"zbrac\" não conceguiu cercar uma
            raiz em',maxiter,' tentativas'
23                  quit()
24              if abs(f(a))<abs(f(b)):
25                  a -= b-a
26                  fa = f(a)
27              else:
28                  b += b-a
29                  fb = f(b)
30
31          erro = abs(b-a)/2.0
32          while erro > tol:
33              contador += 1
34              m = (a+b)/2.0
35              erro /= 2.0
36              fm = f(m)
37              if sinal(fa)*sinal(fm) < 0:
38                  b = m
39              elif sinal(fa)*sinal(fm) > 0:
40                  a = m
41                  fa = fm
42              else:
43                  erro = tol
44                  break
45
46          print 'a raiz é', m
47          print 'com erro menor que', erro
48          print 'em',contador,'iterações'
```

Interpolação Polinomial

```
1           # programa lagrange.py
2
3           xc = (2.0, 3.0, 4.0)
4           yc = (2.0, 3.0, 5.0)
5           n = 3
6
7           x = 3.5
8
9           y = 0
10          for i in range(n):
11              P = 1
12              for j in range(n):
13                  if j == i :continue
14                  P *= (x-xc[j])/(xc[i]-xc[j])
15              y += yc[i]*P
16
17          print 'Para x =',x,' y =',y
```

Integração Numérica

```
1           # programa trapezios1.py
2           # --*-- coding: utf-8 --*--
3
4           # sem guardar os valores intermediários
5
6           # Dados iniciais
7           f = lambda x: x*x
8           a = 0.0
9           b = 1.0
10          n = 4
11
12          # O método dos Trapézios
13          h = (b-a)/n
14          S = 0.0
15          for i in range(1,n):
16              x = a + h*i
17              S += f(x)
18          I = h/2*(f(a)+2*S+f(b))
19
20          print 'A integral é',I
```

```
1           # programa trapezios2.py
2           # --*-- coding: utf-8 --*--
3
4           # sem quardar os valores intermediários
5
6           # Dados iniciais
7           f = lambda x: x*x
8           a = 0.0
9           b = 1.0
10          n = 4
11
12          # O método dos Trapézios
13          S2 = 0.0
14          h = (b-a)/n
15          fa = f(a)
16          f0 = fa
17          f1 = f(a+h)
18          S = f1
19          for i in range(2,n):
20              x = a + h*i
21              f2 = f(x)
22              S += f2
23              temp = abs(f2-2*f1+f0)
24              f0,f1 = f1,f2
25              if temp>S2: S2 = temp
26          I = h/2*(fa+2*S+f(b))
27
28          print 'A integral é',I
29          if n > 2:
30              print 'Com erro menor que',(b-a)*S2/12
```

```
1           # programa simpson.py
2           # --*-- coding: utf-8 --*--
3
4           # sem quardar os valores intermediários
5
6           # Dados iniciais
7           f = lambda x: x*x
8           a = 0.0
9           b = 1.0
10          n = 4
11
12          # Se n for impar acrescentar 1 para torná-lo par
13          if n%2==1: n += 1
14
15          # O método de Simpson
16          h = (b-a)/n
```

```
17          Simpar = 0.0
18          for i in range(1,n,2):
19              x = a + h*i
20              Simpar += f(x)
21          Spar = 0.0
22          for i in range(2,n-1,2):
23              x = a + h*i
24              Spar += f(x)
25
26          I = h/3*(f(a)+4*Simpar+2*Spar+f(b))
```

Problemas de Valor Inicial

O método de Euler

```
1           # programa euler1.py
2           # --*-- coding: utf-8 --*--
3
4           # sem quardar os valores intermediários
5
6           # Dados iniciais
7           f = lambda x,y: x-y
8
9           # passo
10          h = 0.1
11
12          # condições iniciais
13          x = 0.0
14          y = 1.0
15
16          xMax = 2.0
17          print 'x \t y'
18          print x,'\t',y
19          while x < xMax:
20              y = y + h*f(x,y)
21              x += h
22              print x,'\t',y
```

```
1           # programa eulerPC.py
2           # --*-- coding: utf-8 --*--
3
4           # sem quardar os valores intermediários
5
6           # Dados iniciais
```

```
7           f = lambda x,y: x-y
8
9           # passo
10          h = 0.1
11
12          # condições iniciais
13          x = 0.0
14          y = 1.0
15
16          xMax = 2.0
17          print 'x \t y'
18          print x,'\t',y
19          while x < xMax:
20              yp = y + h*f(x,y)
21              x += h
22              y  = y + h*f(x,yp)
23              print x,'\t',y
```

O Método de Runge-Kutta

```
1           # programa RungeKutta.py
2           # coding: utf-8
3
4           # Dados iniciais
5           f = lambda x,y: x-y
6
7           # passo
8           h = 0.1
9
10          # condições iniciais
11          x = 0.0
12          y = 1.0
13
14
15          xMax = 2.0
16          print 'x \t y'
17          print x,'\t',y
18          while x < xMax:
19              k1 = h*f(x,y)
20              k2 = h*f(x+h/2,y+k1/2)
21              k3 = h*f(x+h/2.0,y+k2/2.0)
22              k4 = h*f(x+h,y+k3)
23              y = y + 1.0/6*(k1+2*k2+2*k3+k4)
24              x += h
25              print x,'\t',y
```