Introdução à Programação

Uma Abordagem Funcional utilizando a linguagem Python

Originalmente elaborada por:

Crediné Silva de Menezes,
Maria Claudia Silva Boeres,
Maria Christina Valle Rauber,
Thais Helena Castro,
Alberto Nogueira de Castro Júnior,
Cláudia Galarda Varassin

Adaptada para Python por:

Veruska Carretta Zamborlini

Departamento de Informática - UFES
Departamento de Ciência da Computação – UFAM

2011

Índice

1. CONCEITOS BÁSICOS

1.1. Introdução

Neste curso o leitor estará se envolvendo com a aprendizagem de **conceitos e métodos básicos** para a construção de **programas de computador**. A abordagem que daremos está voltada para o envolvimento do aprendiz com a **solução de problemas** ao invés da atitude passiva de ver o que os outros fizeram. Uma questão central que permeia o curso é a de que construir programas é uma **tarefa de engenharia**, e que, portanto produzirá **artefatos** com os quais o ser humano terá de conviver. Artefatos estes que devem satisfazer **requisitos de qualidade** e serem, portanto, passíveis de constatação.

Optamos desenvolver a disciplina orientada à **descrição de funções**, um formalismo bastante conhecido por todos os que chegam a este curso. Esperamos, com isso, atenuar algumas dificuldades típicas do ensino introdutório de programação. Nas seções seguintes apresentamos alguns conceitos básicos, importantes de se ter em mente antes de iniciarmos um curso de programação.

1.2. Computadores

Denominamos **computador** uma máquina de processar dados, numéricos ou simbólicos, que funciona através da execução de programas. Ao contrário das inúmeras máquinas que conhecemos, tais como máquina de lavar roupa, liquidificador, aspirador de pó, e tantas outras, que realizam uma única função, o computador é uma máquina multi-uso. Podemos usá-lo como uma máquina de escrever sofisticada, como uma máquina de fax, como uma prancheta de desenho sofisticada, como um fichário eletrônico, como uma planilha de cálculos e de tantas outras formas. É exatamente como o nosso conhecido videogame: para mudar de jogo basta trocar o cartucho. No videogame, cada novo jogo é determinado por um novo programa.

Em linhas gerais podemos entender um computador como uma máquina capaz de:

a) interpretar dados que lhe são fornecidos, produzindo resultados em forma de novos dados, usando para isso conceitos que lhe foram antecipadamente informados e,
b) aceitar a descrição de novos conceitos e considerá-los na interpretação de novas situações.

Alguns exemplos de uso de um computador:

1) Descrever para uma máquina a relação métrica que existe entre os lados de um triângulo retângulo. De posse desse conhecimento, a máquina poderia, por exemplo, determinar o valor de um dos lados quando conhecido o valor dos outros dois.

2) Informar a uma máquina as regras de conjugação de verbos. Com este conhecimento a máquina pode determinar a forma correta para um determinado tempo e pessoa de um verbo específico.

3) Tradução de textos;

4) Classificação de textos quanto à sua natureza: romance, poesia, documentário, entrevista, artigo científico;
5) Manipulação de expressões algébricas, resolução de integral indefinida, etc;
6) Programação automática: dada uma especificação, gerar um programa eficiente;
7) Monitoramento de pacientes em um Centro de Tratamento Intensivo;
8) Identificação de tumores no cérebro a partir da comparação de imagens com padrões conhecidos de anormalidade;
9) Roteamento inteligente de mensagens;
10) Monitoramento de regiões por satélite.

1.3. Programação

À tarefa de identificar o conhecimento necessário para a descrição de um conceito, organizá-lo e codificá-lo de modo a ser entendido pela máquina damos o nome de ***programação de computadores***. Ao conhecimento codificado, produto final da tarefa de programação dá-se o nome de ***programa***.

A programação de computadores é uma atividade que compreende várias outras atividades, tais como: entendimento do problema a ser resolvido, planejamento de uma solução, formalização da solução usando uma linguagem de programação, verificação da conformidade da solução obtida com o problema proposto.

1.4. Linguagem de Programação

A descrição de conhecimento para um agente racional qualquer (seja uma máquina ou um humano) subentende a existência de padrões segundo os quais o agente possa interpretar o conhecimento informado. A esses padrões, quando rigorosamente elaborados, damos o nome de *formalismo*. Um formalismo é composto de dois aspectos: a sintaxe e a semântica.

A sintaxe permite ao agente reconhecer quando uma "sequência de símbolos" que lhe é fornecida está de acordo com as regras de escrita e, portanto representa um programa. A semântica permite que o agente atribua um significado ao conhecimento descrito pela "sequência de símbolos". Por exemplo, quando um agente humano (com determinado grau de escolaridade) encontra a seguinte sequência de símbolos {3, 4} U {5, 9, 15}, ele por certo reconhecerá como uma expressão algébrica escrita corretamente e, se lembrar dos fundamentos da teoria dos conjuntos, associará esta cadeia como a descrição de um conjunto composto pela união dos elementos de dois conjuntos menores.

Eis aqui algumas observações importantes sobre a necessidade de linguagens de programação:

- Ainda não é possível usar linguagem natural para ensinar o computador a realizar uma determinada tarefa. A linguagem natural, tão simples para os humanos, possui ambiguidades e redundâncias que a inviabilizam como veículo de comunicação com os computadores.

- A linguagem nativa dos computadores é muito difícil de ser usada, pois requer do programador a preocupação com muitos detalhes específicos da máquina, tirando pois atenção do problema.
- Para facilitar a tarefa de programação foram inventadas as linguagens de programação. Estas linguagens têm por objetivo se colocarem mais próximas do linguajar dos problemas do que do computador em si. Para que o programa que escrevemos possa ser “entendido” pelo computador, existem programas especiais que os traduzem (compiladores) ou os que interpretam (interpretadores) para a linguagem do computador.
- Podemos fazer um paralelo com o que ocorre quando queremos nos comunicar com uma pessoa de língua estrangeira. Podemos escrever uma carta em nossa língua e pedir a alguém que a traduza para a língua de nosso destinatário ou se quisermos conversar pessoalmente, podemos usar um intérprete.

1.5. Propriedades de um Programa

Fazemos programas com a intenção de dotar uma máquina da capacidade de resolver problemas. Neste sentido, um programa é um produto bem definido, que para ser usado precisa que sejam garantidas algumas propriedades. Aqui fazemos referências a duas delas: a correção e o desempenho. A correção pode ser entendida como a propriedade que assegura que o programa descreve corretamente o conhecimento que tínhamos intenção de descrever. O desempenho trata da propriedade que assegura que o programa usará de forma apropriada o tempo e os recursos da máquina considerada. Cabe aqui alertar aos principiantes que a tarefa de garantir que um programa foi desenvolvido corretamente é tão complexa quanto à própria construção do programa em si. Garantir que um programa funciona corretamente é condição imprescindível para o seu uso e, portanto estaremos dando maior ênfase a esta propriedade.

1.6. Paradigmas de Linguagens de Programação

As regras que permitem a associação de significados às "sequências de símbolos" obedecem a certos princípios. Existem várias manifestações destes princípios e a cada uma delas denominamos de paradigma.

Um paradigma pode ser entendido informalmente como uma forma específica de se "pensar" sobre programação. Existem três grandes grupos de paradigmas para programação: o procedimental (ou procedural), o funcional e o lógico. Os dois últimos são frequentemente referidos como sendo subparadigmas de um outro mais geral, o paradigma declarativo. O paradigma procedimental subentende a organização do conhecimento como uma sequência de tarefas para uma máquina específica. O paradigma lógico requer o conhecimento de um formalismo matemático denominado lógica matemática. O paradigma funcional baseia-se no uso dos princípios das funções matemáticas. De uma forma geral, os paradigmas declarativos enfatizam o aspecto correção e o procedimental os aspectos de desempenho. Vejam que falamos em "enfatizam", o que quer dizer que apresentam facilidades para descrição e verificação da propriedade considerada. Entretanto, em qualquer caso, o programador deverá sempre garantir que os dois aspectos (correção e desempenho) sejam atendidos.

1.7. Programação Funcional

Para os fins que aqui nos interessam neste primeiro momento, podemos entender o computador, de uma forma simplificada, como uma máquina capaz de:

a) avaliar expressões escritas segundo regras sintáticas bem definidas, como a das expressões aritméticas que tão bem conhecemos (ex. 3 + 5 - 8) obedecendo à semântica das funções primitivas das quais ela é dotada (por exemplo: as funções aritméticas básicas como somar, subtrair, multiplicar e dividir);
b) aceitar a definição de novas funções e posteriormente considerá-las na avaliação de expressões submetidas à sua avaliação.

Por enquanto, denominaremos o computador de máquina funcional. Na Figura 1.1 apresentamos um exemplo de interação de um usuário com a nossa Máquina Funcional.

usuário:	6 + 4 / 2
Máquina funcional::	8
usuário:	f(x, y) = (x + y) / 2
Máquina funcional:	definição de **f** foi aceita
usuário:	f(6, 4) + f(10, 40)
Máquina funcional:	30

Figura 1.1

Na primeira interação podemos observar que o usuário descreveu uma expressão aritmética e que a máquina funcional avaliou e informou o resultado. Na segunda interação o usuário descreve, através de uma equação, uma nova função, que ele denominou de **f** e que a máquina funcional acatou a nova definição. Na terceira interação o usuário solicita a avaliação de uma nova expressão aritmética usando o conceito recentemente definido e que a máquina funcional faz a avaliação usando corretamente o novo conceito. Desta forma, percebemos que a máquina funcional é capaz de avaliar expressões aritméticas e funções e também aceitar definições de funções, usando para isso, ambientes distintos.

1.8. Expressões Aritméticas

A nossa máquina funcional hipotética entende a sintaxe das expressões aritméticas, com as quais todo aluno universitário já é bem familiarizado e é capaz de avaliá-las usando as mesmas que regras que já conhecemos.

Sintaxe - Todo operador aritmético pode ser entendido, e aqui o será, como uma função que possui dois parâmetros. A notação usual para as operações aritméticas é a infixada, ou seja, símbolo funcional colocado entre os dois operandos. Nada impede de pensarmos nelas escritas na forma prefixada, que é a notação usual para funções com número de parâmetros diferente de 2. Por exemplo, podemos escrever "***+ 3 2***" para descrever a soma do número 3 com o número 2. As funções definidas pelo programador devem ser escritas de forma prefixada, como no exemplo de interação apresentado na Figura 1.1. Combinando essas duas formas, infixada e prefixada, podemos escrever expressões bastante sofisticadas.

Avaliação - As expressões aritméticas, como sabemos, são avaliadas de acordo com regras de avaliação bem definidas, efetuando as operações de acordo com suas prioridades. Por exemplo, na expressão "***6 + 4 / 2***" o primeiro operador a ser avaliado será o de divisão (/) e posteriormente o de adição. Se desejarmos mudar essa ordem, podemos usar parêntesis em qualquer quantidade, desde que balanceados e em posições apropriadas. Por exemplo, na expressão "***(6 + 4) / 2***", a utilização de parêntesis determina que a sub-expressão 6 + 4 terá prioridade na avaliação.

1.9. Funções

Podemos entender o conceito de funções como uma associação entre elementos de dois conjuntos A e B de tal forma que para cada elemento de A existe apenas um elemento de B associado. O conjunto A é conhecido como o domínio da função, ou ainda como o conjunto de entrada, e o conjunto B é o contra-domínio ou conjunto de saída. Para ser mais preciso, podemos afirmar que uma função *f*, que associa os elementos de um conjunto A aos elementos de um conjunto B, consiste em um conjunto de pares ordenados onde o primeiro elemento do par pertence a A o segundo a B. Exemplos:

a) Seja a função T que associa as vogais do alfabeto com os cinco primeiros inteiros positivos.

   T = {(a,1), (e,2), (i,3), (o,4), (u,5)}

b) Seja a função Q, que associa a cada número natural o seu quadrado.

   Q = {(0,0), (1,1), (2,4), (3,9), (4,16), ...}

Podemos observar que a função T é um conjunto finito e que a função Q é um conjunto infinito.

1.10. Descrições Funcionais

Podemos descrever um conjunto, de duas formas: *extensional*, onde explicitamos todos os elementos que são membros do conjunto, como no caso do conjunto T apresentado anteriormente; ou na forma *intencional*, onde descrevemos um critério de pertinência dos membros do conjunto. Por exemplo, o conjunto Q acima apresentado poderia ser reescrito da seguinte forma:

Q = {(x, y) | x é natural e y = x.x} que pode ser lido da seguinte maneira:

Q é o conjunto dos pares ordenados (x, y) tal que x é um número natural e y é o produto de x por x.

Quando descrevemos uma função para fins computacionais, estamos interessados em explicitar como determinar o segundo elemento do par ordenado, conhecido o primeiro elemento do par. Em outras palavras, como determinar y conhecendo-se o valor de x. Normalmente dizemos que queremos determinar y em função de x. Nesta forma de descrição, omitimos a variável y e explicitamos o primeiro elemento que é denominado então de parâmetro da função. No caso acima teríamos então:

Q(x) = x . x

1.11. Por que Começar a Aprendizagem de Programação através do Paradigma Funcional?

Tendo em vista a prática vigente de começar o ensino de programação em cursos de computação utilizando o paradigma procedimental, apresentamos a seguir alguns elementos que baseiam nossa opção de começar o ensino de programação usando o paradigma funcional.

1) O aluno de graduação em Computação tem de 4 a 5 anos para aprender todos os detalhes da área de computação, portanto não se justifica que tudo tenha que ser absorvido no primeiro semestre. O curso introdutório é apenas o primeiro passo e não visa formar completamente um programador. Este é o momento de apresentar-lhe os fundamentos e, além disso, permitir que ele vislumbre a variedade de problemas que podem ser solucionados como o apoio do computador;
2) O paradigma procedimental requer que o aluno tenha um bom entendimento dos princípios de funcionamento de um computador real, pois eles se baseiam, como as máquinas reais, no conceito de mudança de estados (máquina de Von Neumann).
3) O paradigma lógico, outro forte candidato, requer o conhecimento de lógica matemática que o aluno ainda não domina adequadamente;
4) O paradigma funcional é baseado num conhecimento que o aluno já está familiarizado desde o ensino médio (funções, mapeamento entre domínios) o qual é ainda explorado em outras disciplinas do ciclo básico, o que nos permite concentrar nossa atenção na elaboração de soluções e na descrição formal destas;
5) O elevado poder de expressão das linguagens funcionais permite que a abrangência do uso do computador seja percebida mais rapidamente. Em outras palavras, podemos resolver problemas mais complexos já no primeiro curso;
6) O poder computacional do paradigma funcional é idêntico ao dos outros paradigmas. Apesar disso, ele ainda não é usado nas empresas, por vários aspectos. Dentre os quais podemos citar:
    a. No passado, programas escritos em linguagens funcionais eram executados muito lentamente. Apesar disso não ser mais verdadeiro, ficou a fama;
    b. A cultura de linguagens procedimentais possui muitos adeptos no mundo inteiro, o que, inevitavelmente, cria uma barreira à introdução de um novo paradigma. Afinal, temos medo do desconhecido e trememos quando temos que nos livrar de algo que já sabemos;
    c. Há uma crença que linguagens funcionais são difíceis de aprender e só servem para construir programas de inteligência artificial.
7) A ineficiência das linguagens funcionais em comparação às procedimentais tem se reduzido através de alguns mecanismos tais como: lazy evaluation, grafo de redução, combinadores.
8) Para fazer um programa que "funciona" (faz alguma coisa, embora não necessariamente o que desejamos) é mais fácil fazê-lo no paradigma procedimental. Para fazer um programa que funciona "corretamente" para resolver um determinado

problema é mais fácil no paradigma funcional, pois esse paradigma descreve "o que fazer" e não "como fazer".

9) As linguagens funcionais são geralmente utilizadas para processamento simbólico, ou seja, solução de problemas não numéricos. (Exemplo: integração simbólica X integração numérica). Atualmente constata-se um crescimento acentuado do uso deste tipo de processamento.

10) A crescente difusão do uso do computador nas mais diversas áreas do conhecimento gera um crescimento na demanda de produção de programas cada vez mais complexos. Os defensores da programação funcional acreditam que o uso deste paradigma seja uma boa resposta a este problema, visto que com linguagens funcionais podemos nos concentrar mais na solução do problema do que nos detalhes de um computador específico, o que aumenta a produtividade.

11) Se mais nada justificar o aprendizado de uma linguagem funcional como primeira linguagem, resta a explicação didática. Estamos dando um primeiro passo no entendimento de programação como um todo. É, portanto, importante que este passo seja simplificado através do apoio em uma "máquina" já conhecida, como é o caso das funções.

12) Ainda um outro argumento: mesmo que tenhamos que usar posteriormente uma outra linguagem para ter uma implementação eficiente, podemos usar o paradigma funcional para formalizar a solução. Sendo assim, a versão em linguagem funcional poderia servir como uma especificação do programa.

EXERCÍCIOS

1. Conceitue programação de computadores.
2. Quais os principais paradigmas de programação e o que os diferenciam.
3. Faça uma descrição intencional da função: F = {1,3,5,7,...}.
4. Faça uma listagem de outros exemplos de programas de computador que são usados hoje em diferentes áreas do conhecimento e por diferentes profissionais.
5. Apresente exemplo de outras linguagens técnicas usadas pelo ser humano para descrever conhecimento.
6. Os conceitos de correção e de desempenho, se aplicam a qualquer artefato. Escolha 3 artefatos quaisquer e discuta os dois conceitos.
7. Apresente uma descrição informal da função que conjuga os verbos regulares da 1a. conjugação.

2. A LINGUAGEM DE PROGRAMAÇÃO PYTHON E O AMBIENTE IDLE

2.1. Introdução

Neste curso usaremos a linguagem Python de programação através do ambiente IDLE. Embora a linguagem seja multiparadigmada, abrangendo os paradigmas procedural, orientação a objeto e funcional, ela será usada neste curso como ferramenta para aprendizado de noções de programação através deste último paradigma. De fato, tal linguagem de programação apresenta estruturas de dados de alto nível poderosas e eficientes, embora possua uma sintaxe simples e elegante, de fácil aprendizado.

Podemos usar o IDLE como uma calculadora qualquer, à qual submetemos expressões que ela avalia e nos informa o valor resultante. Vejamos por exemplo as interações a seguir.

```
>>> 3 * 5
15
>>> 6 + 4/2
8
>>> (6 + 4)/2
5
>>>
```

Nas expressões acima é importante destacar o uso do símbolo * (asterisco) que é empregado para representar a multiplicação. Além desse, outros símbolos usuais serão substituídos, como veremos logo mais.

No ambiente IDLE, o símbolo **>>>** é usado para indicar que o sistema está preparado para avaliar uma nova expressão. Após avaliar uma expressão o resultado é informado na linha seguinte e em seguida tal símbolo é novamente exibido, se disponibilizando para uma nova avaliação. O avaliador de expressões do ambiente IDLE funciona conforme o esquema da figura 2.1. Repetidamente o ambiente IDLE permite a leitura de uma expressão, sua avaliação e exibição do resultado.

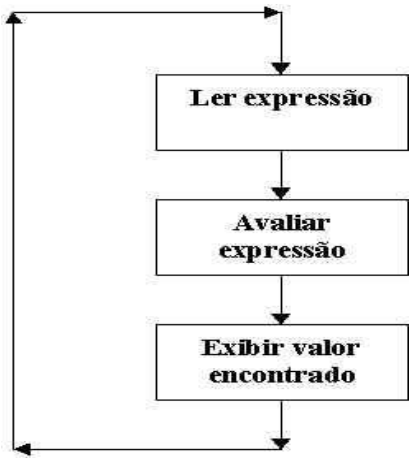


Figura 2.1

Expressões são aplicações de funções que manipulam valores para produzir um resultado. Por exemplo, a expressão **`3 * 5`** é a aplicação da função de multiplicação para manipular os valores 3 e 5 e produzir o resultado 15. As operações aritméticas são, como sabemos, funções matemáticas. A linguagem Python oferece operações aritméticas básicas utilizando a notação usual, ou seja, infixada (o símbolo da operação fica entre os operandos). Elas são ditas funções **primitivas**, ou seja, funções já oferecidas pelo ambiente.

Além de usarmos tal ambiente para escrever expressões e solicitar ao sistema que as avalie, podemos ousar mais e usá-lo também para descrever novas funções. Podemos utilizar nas definições tanto funções primitvas quanto outras funções que tivermos construído anteriomente.

2.2. Descrição de Funções

Podemos definir uma função que determina o espaço percorrido por um móvel em movimento retilíneo uniforme (MRU), conhecidos a sua velocidade e o tempo decorrido. Podemos descrever esta função através de uma equação matemática, conforme discutido no Capítulo 1, e também definí-la na linguagem Python:

Notação Matemática	***Notação de Pyhton***
espaco(v, t) = v x t	**`def`** `espaco(v, t): return v * t`

No esquema a seguir fazemos uma identificação didática dos vários elementos da definição. O lado esquerdo da função é chamado de inteface ou assinatura, enquanto o lado direito é chamado de corpo da definição.

	INTERFACE DA FUNÇÃO			**CORPO DA DEFINIÇÃO**
	nome da função	parâmetros		expressão aritmética que define a relação que há entre os parâmetros
`def`	`espaco`	`(v, t)`	**`: return`**	`v * t`

No lado esquerdo, temos a palavra reservada **`def`**, imediatamente seguida do nome dado à função (**`espaco`**) e dos parâmetros (ou argumentos) da função entre parêntesis, separados por vírgula. O símbolo “dois pontos”, que corresponde ao sinal de igual da notação matemática, separa a interface da função da sua definição. No lado direito, escrevemos uma expressão utilizando outras funções, primitivas ou não. Tal expressão é precedida da palavra reservada **`return`** que, intuitivamente, significa que o valor calculado da expressão será informado (retornado) ao ambiente para ser exibido como resultado.

Assim, podemos alimentar o ambiente IDLE com uma função para calcular o espaço percorrido por um móvel em MRU, e então usá-la tantas vezes quanto necessário.

```
>>> def espaco(v, t): return v * t                    [Enter 2x]
>>> espaco(3, 5)
15
>>> espaco(10, 20)
200
```

No ambiente IDLE, deve-se teclar ‘Enter’ duas vezes ao terminar a definição de uma função.

Observe que uma função definida sem parâmetros é tida como uma constante. Por exemplo:

```
>>> def pi(): return 3.1416                                    [Enter 2x]
>>> pi()
3.1415999999999999
```

Portanto, na nomenclatura de definições de funções, dizemos que temos definições **paramétricas** e definições **não-paramétricas** ou **constantes**.

2.3. Definições Globais e Locais

As definições que discutimos até então são chamadas **globais**, no sentido de que estão acessíveis ao uso direto do usuário e também disponíveis para uso em qualquer outra definição. Por exemplo, se definimos:

```
>>> def quad(x): return x * x                                  [Enter 2x]
>>> def pi(): return 3.1416                                    [Enter 2x]
```

podemos utilizar tais funções tanto externamente:

```
>>> quad(4)
16
>>> pi()
3.1415999999999999
```

quanto internamente a outra definição:

```
>>> def areaCirc(r): return pi() * quad(r)                     [Enter 2x]
>>> areaCirc(3)
28.4526
```

Se, no entanto, desejarmos construir subdefinições para uso em uma definição específica, podemos definir funções **locais**, ou seja, funções internas a outras funções. Neste caso, o contexto de aplicação da função local é restrito ao escopo da função em que ela é definida. Por exemplo, definimos a função **`quad2`** e **`pi2`** internamente à função **`areaCirc2`**:

```
>>> def areaCirc2(r):
        def pi2(): return 3.1614
        def quad2(): return r * r
        return pi2() * quad2()                                 [Enter 2x]
>>> areaCirc2(3)
28.4526
```

O resultado obtido é o mesmo, porém, neste caso, as funções locais **`quad2`** e **`pi2`** não são acessíveis ao uso direto do usuário. Se tentarmos utilizar tais funções externamente, o ambiente retornará um erro, pois elas são válidas apenas internamente ao escopo da função **`areaCirc2`**:

```
>>> quad2(3)
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    quad2(3)
NameError: name 'quad2' is not defined
```

Além disso, as definições internas podem ou não ser paramétricas. Neste exemplo, utilizamos funções **locais não-paramétricas**. Note que, apesar de ***r*** não ser parâmetro da função **`quad2`**, ele foi utilizado em sua definição. Isto é possível porque ***r*** é parâmetro da função **`areaCirc2`** e tem validade em todo o lado direito de sua definição.

Observe ainda que a **identação** em Python tem significado, pois é assim que se determina o que está dentro do escopo de cada função. Por exemplo, a mesma definição da função **`areaCirc2`** sem o uso da identação provoca o erro sintático "espera-se um bloco identado", como mostrado a seguir:

```
>>> def areaCirc2(r):
def pi2(): return 3.1614
def quad2(): return r * r
return pi2() * quad2()                          [Enter 2x]

  File "<pyshell#33>", line 2
    def pi2(): return 3.1614
      ^
IndentationError: expected an indented block
```

Ou seja, a identação determina que as expressões **`pi2`**, **`quad2`** e **`return`** estão dentro do escopo da função **`areaCirc2`**.

Ademais, temos que considerar que nada impede a definição de funções locais aninhadas uma dentro da outra, paramétricas ou não. Por exemplo, veja a função a seguir que calcula a média de dois números:

```
>>> def f(x, y):
       def h():
            def g(x): return x/2
            return g(x) + g(y)
       return h()                                  [Enter 2x]
>>> f(4,8)
6
```

É importante observar que, ao contrário do que acontece no exemplo de **`areaCirc2`**, a função **local paramétrica `g`** tem um parâmetro **`x`** que nada tem a ver com o parâmetro **`x`** da função **`f`**. Esta situação é análoga à composição de funções matemáticas. Se temos f(x) = 3x e g(x) = x^2, a composta f(g(2)) = 12 é resolvida como g(2) = 4 e f(g(2)) = f(4) = 12, ou seja, o fato de **f** e **g** serem definidas em termos de um argumento com mesmo nome, não implica que elas serão resolvidas para um mesmo valor. Por outro lado, os valores **`x`** e **`y`** manipulados dentro da função **local não-paramétrica `h`** são, assim como no exemplo da área do círculo, os mesmos passados como parâmetro para a função **`f`**.

2.4. Modelo de Avaliação de Expressões:

Quando o interpretador do IDLE toma uma expressão aritmética para avaliar, ele efetua as operações obedecendo à prioridade dos operadores, bem como ao escopo determinado pelo uso de parêntesis. Por exemplo, a avaliação das expressões 4 + 6 / 2 e (4 + 6) / 2 resulta em valores diferentes. Para ilustrar a diferença, vejamos a seguir dois esquemas de avaliação dessas expressões como uma sequência de reduções à expressões cada vez

mais simples, até que se atinja um termo irredutível (a seta ➔ é usada para indicar um passo de redução):

```
 4 + 6 / 2 ➔ 4 + 3 ➔ 7
(4 + 6) / 2 ➔ 10 / 2 ➔ 5
```

As expressões que utilizam funções definidas pelo usuário são avaliadas por um processo análogo. Cada referência a uma função é substituída por sua definição, ou seja, seu lado direito, até que se atinja uma expressão básica, prosseguindo-se então como no caso acima. Vejamos a seguir um exemplo de avaliação da soma das áreas de duas circunferências, segundo a definição **`areaCirc`** apresentada anteriormente:

ordem	**expressão**	**redução a ser aplicada**
1	**`areaCirc(3) + areaCirc(5)`**	definição de **`areaCirc`**
2	**`(pi() * quad(3)) + areaCirc(5)`**	definição de **`PI`**
3	**`(3.1416 * quad(3)) + areaCirc(5)`**	definição de **`quad`**
4	**`(3.1416 * 9) + areaCirc(5)`**	definição de **`areaCirc`**
5	**`(3.1416 * 9) + (pi() * quad(5))`**	definição de **`pi`**
6	**`(3.1416 * 9) + (3.1416 * quad(5))`**	definição de **`quad`**
7	**`(3.1416 * 9) + (3.1416 * 25)`**	operação *
8	**`28.4526 + (3.1416 * 25)`**	operação *
9	**`28.4526 + 79.035`**	operação +
10	**`107.4876`**	

Para uma realização específica da linguagem, isso não precisa acontecer exatamente nessa ordem. Entretanto este modelo é suficiente para nossos interesses. O número de reduções necessárias para chegar à forma irredutível de uma expressão, também denominada de forma canônica ou ainda forma normal, pode ser usado como critério para discutir o desempenho da mesma.

As relações de precendência dos operadores matemáticos serão discutidas mais a frente. Por ora, basta sabermos a precedência dos operadores básicos, semelhante à que estudamos classicamente em matemática: operações de multiplicação (“*”), divisão (“/”), e resto(“%”) tem precedência sobre soma (“+”) e subtração (“-“).

2.5. Arquivo-fonte ou Script

As definições de funções realizadas no ambiente IDLE são temporárias, ou seja, se ele for reiniciado as definições serão perdidas. Assim, uma alternativa a isto é a criação de um **arquivo-fonte** ou **script** com as definições, de forma que essas sejam persistidas e possam ser reutilizadas sempre que necessário. Cada arquivo pode ter uma ou mais definições de funções. Normalmente agrupamos em um mesmo arquivo as definições relacionadas com a solução de um problema específico ou definições de propósito geral.

Uma maneira simples de criar e usar um arquivo-fonte em Python é usando o próprio ambiente IDLE. No menu *File* escolha a opção *New Window*. Será aberta uma nova janela em branco, em que se pode escrever as definições de funções. Observe que estas

definições não são cópia integral do ambiente IDLE. Um exemplo de definição de função em arquivo é:

```
def f(x, y):
      def h():
            def g(x): return x/2
            return g(x) + g(y)
      return h()

def d(x):
      return x*x
```

Uma vez codificadas as funções desejadas, deve-se salvar o arquivo (menu *File*, opção *Save*), para então poder ser carregado. Para tanto, escolha no menu *Run* a opção *Run Module*. Se não houver erro no código, o interpretador do IDLE será alimentado com as definições de funções contidas no arquivo.

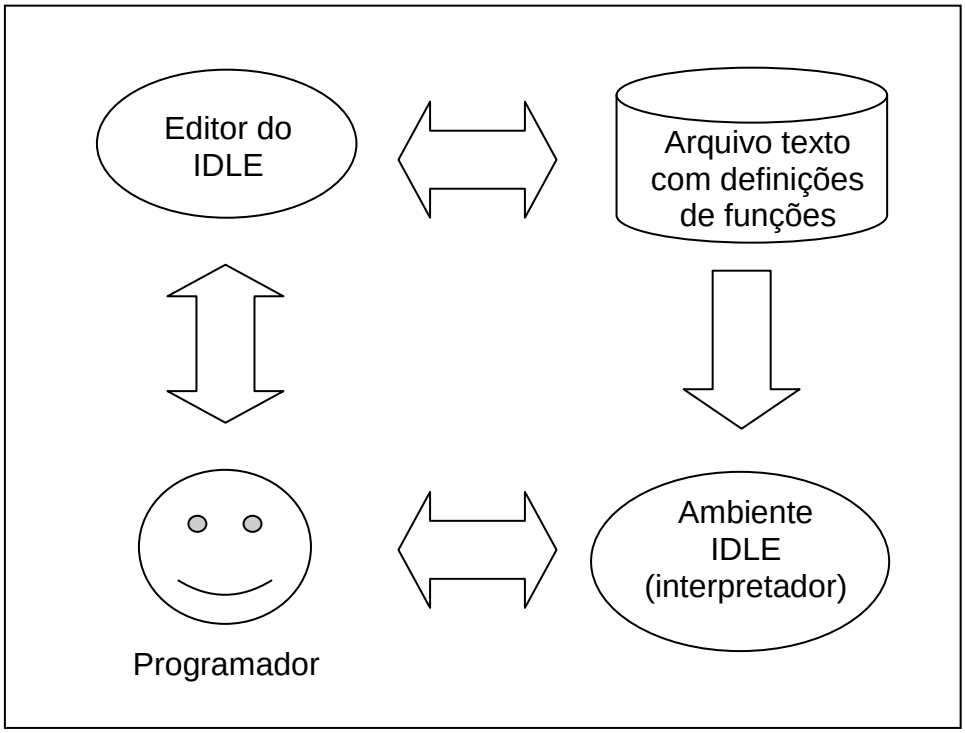


Figura 2.2 – Interações do Programador com o Ambiente de Programação

2.6. Uso de Bibliotecas/Módulos

As expressões e funções que estudamos até agora foram definidas em termos de operadores aritméticos. Como vimos, estes operadores são funções primitivas oferecidas pelo ambiente, ou seja, não precisam ser definidos. Outras funções, inclusive matemáticas, estão disponíveis em bibliotecas ou módulos que podem ser importados, ou seja, inseridos no ambiente IDLE, conforme a necessidade.

Por exemplo, o módulo **`math`** oferece acesso às funções da biblioteca C para matemática. A seguir apresentamos duas formas de importar todas as funções definidas em um módulo.

```
```

```
>>> import math
>>> math.srqt(4)
2
```

```
>>> from math import *
>>> srqt(4)
2
```

Ou ainda, se desejamos apenas algumas funções em particular, podemos importá-la diretamente da biblioteca:

```
>>> from math import sqrt, sin
>>> srqt(4)
2
```

À medida que for necessário, em aula ou para exercícios e trabalhos, introduziremos outras bibliotecas com as funcionalidades adequadas.

2.7. Um exemplo

Considere que queremos descrever uma função para determinar as raízes de uma equação do segundo grau. Sabemos que pela nossa clássica fórmula as raízes são descritas genericamente por:

$$x = \frac{-b \pm \sqrt{b^2 - 4 \times a \times c}}{2 \times a}$$

A solução, como sabemos, é formada por um par de valores. Por enquanto vamos descrever este fato por duas funções, uma para a primeira raiz e outra para a segunda.

```
>>> from math import sqrt
>>> def raiz1eq2gr(a,b,c): return (-b + sqrt(b*b - 4*a*c))/(2*a)
>>> def raiz2eq2gr(a,b,c): return (-b - sqrt(b*b - 4*a*c))/(2*a)
```

Vamos discutir alguns detalhes desta codificação:

- a raiz quadrada é calculada pela função ***sqrt***, importada da biblioteca ***math***;
- o numerador da fração precisa ser delimitado pelo uso de parêntesis;
- o denominador também precisa ser delimitado por parêntesis;
- o símbolo de multiplicação usual foi trocado pelo * (asterisco);

Visto que as duas descrições possuem partes comuns, poderíamos ter escrito funções auxiliares e produzir um conjunto de definições, tal como:

```
>>> def raiz1eq2gr(a, b, c): return (-b + sqrt(delta(a, b, c)))/dobro(a)
>>> def raiz2eq2gr(a, b, c): return (-b - sqrt(delta(a, b, c)))/dobro(a)
>>> def delta(a, b, c): return quad(b) - 4.0 * a * c
>>> def quad(x): return x * x
>>> def dobro(x): return 2.0 * x
```

Vejamos como ficaria uma interação com o IDLE:

```
>>> raiz1eq2gr(2, 5, 2)
-0.5
>>> raiz2eq2gr(2, 5, 2)
-2.0
>>> raiz1eq2gr(3, 4, 5)

Traceback (most recent call last):
```

```
  File "<pyshell#31>", line 1, in <module>
    raiz1eq2gr(3, 4, 5)
  File "<pyshell#26>", line 1, in raiz1eq2gr
    def raiz1eq2gr(a, b, c): return (-b + sqrt(delta(a, b, c)))/dobro(a)
ValueError: math domain error
```

Podemos observar que houve um problema com a avaliação da expressão

```
raiz1eq2gr(3, 4, 5)
```

Este é um erro semântico, provocado pela tentativa de extrair a raiz quadrada de um número negativo. A função que definimos, portanto, é uma função parcial, ou seja, ela não está definida para todo o domínio dos reais. Neste caso o sistema apenas acusa o problema ocorrido.

EXERCÍCIOS

1. Avalie as expressões abaixo e apresente a sequência de reduções necessárias para a obtenção do termo irredutível (resultado final):
   a. 15 % 2
   b. 15 % 2 + 6 / 3
   c. sqrt (15 – 2*3) / (17 – 12)
2. Escreva um arquivo texto (script) contendo a definição das funções abaixo. Use, quando for adequado, definições locais:
   a. Determinação da área de um retângulo de lados a e b
   b. Determinação da área de um círculo de raio r
   c. Determinação da média aritmética de três números a, b e c
3. Apresente uma explicação para o erro produzido pela avaliação da expressão

   ```
   raiz1eq2gr(3, 4, 5)
   ```

4. Apresente uma sequência de reduções para a expressão:

   ```
   raiz1eq2gr(2, 5, 2)
   ```

3. A ARTE DE RESOLVER PROBLEMAS

3.1. Introdução

O grande objetivo deste curso é criar oportunidades para que o estudante desenvolva suas habilidades como resolvedor de problemas. Mais especificamente estamos interessados na resolução de problemas usando o computador, entretanto, temos certeza, que as idéias são gerais o suficiente para ajudar a resolver qualquer problema.

Embora muitas pessoas já tenham discutido sobre este assunto ao longo da história, nosso principal referencial será o professor George Polya, que escreveu um livro sobre este assunto, com o mesmo nome deste capítulo, voltado para a resolução de problemas de matemática do ensino fundamental. Todos os alunos deste curso terão grande proveito se lerem este livro.

As idéias ali apresentadas com certeza se aplicam com muita propriedade à resolução de problemas com o computador. Na medida do possível estaremos apresentando aqui algumas adaptações que nos parecem apropriadas neste primeiro contato com a resolução de problemas usando o computador.

3.2. Dicas Iniciais

Apresenta-se a seguir algumas orientações:

3.2.1. Só se aprende a resolver problemas através da experiência. Logo o aluno que está interessado em aprender deve trabalhar, buscando resolver por si mesmo, os problemas propostos antes de tentar o auxílio do professor ou de outro colega;

3.2.2. A ajuda do professor não deve vir através da apresentação pura e simples de uma solução. A ajuda deve vir através do fornecimento de pistas que ajudem o aluno a descobrir a solução por si mesmo;

3.2.3. É muito importante não se conformar com uma única solução. Quanto mais soluções encontrar, mais hábil o estudante ficará, e, além disso, poderá comparar as várias alternativas e escolher a que lhe parecer mais apropriada. A escolha deverá sempre ser baseada em critérios objetivos.

3.2.4. Na busca pela solução de um problema, nossa ferramenta principal é o questionamento. Devemos buscar formular questões que nos ajudem a entender o problema e a elaborar a solução.

3.2.5. Aprenda desde cedo a buscar um aprimoramento da sua técnica para resolver problemas, crie uma sistematização, evite chegar na frente de um problema como se nunca tivesse resolvido qualquer outro problema. Construa um processo individual e vá aperfeiçoando-o a cada vez que resolver um novo problema.

3.3 Como Resolver um Problema

O professor Polya descreve a resolução de um problema como um processo complexo que se divide em quatro etapas, as quais apresentamos aqui, com alguma adaptação. Segundo

nos recomenda Polya, em qualquer destas etapas devemos ter em mente três questões sobre o andamento do nosso trabalho, que são: Por onde começar esta etapa? O que posso fazer com os elementos que disponho? Qual a vantagem de proceder da forma escolhida?

Etapa 1 - Compreensão do Problema

É impossível resolver um problema sobre o qual não tenhamos um entendimento adequado. Portanto, antes de correr atrás de uma solução, concentre-se um pouco em identificar os elementos do problema. Faça perguntas tais como: Quais são os dados de entrada? O que desejamos produzir como resultado? Qual a relação que existe entre os dados de entrada e o resultado a ser produzido? Quais são as propriedades importantes que os dados de entrada possuem?

Etapa 2 – Planejamento

Nesta fase devemos nos envolver com a busca de uma solução para o problema proposto. Pode ser que já o tenhamos resolvido antes, para dados ligeiramente modificados. Se não é o caso, pode ser que já tenhamos resolvido um problema parecido. Qual a relação que existe entre este problema que temos e um outro problema já conhecido? Será que podemos quebrar o problema em problemas menores? Será que generalizando o problema não chegaremos a um outro já conhecido? Conhecemos um problema parecido, embora mais simples, o qual quando generalizado se aproxima do que temos?

Etapa 3 - Desenvolvimento

Escolhida uma estratégia para atacar o problema, devemos então caminhar para a construção da solução. Nesta fase, devemos considerar os elementos da linguagem de programação que iremos usar, respeitando os elementos disponibilizados pela linguagem, tais como tipos, formas de definição, possibilidades de generalização e uso de elementos anteriormente definidos. A seguir, devemos codificar cada pedaço da solução, e garantir que cada um esteja descrito de forma apropriada. Em outras palavras, não basta construir, temos que ser capazes de verificar se o resultado de nossa construção está correto. Isto pode ser realizado pela identificação de instâncias típicas, da determinação dos valores esperados por estas, da submissão dessas instâncias ao avaliador e finalmente, da comparação dos resultados esperados com os resultados produzidos pela avaliação de nossas definições. Além disso, devemos garantir que a definição principal também está correta.

Em síntese, a fase de desenvolvimento compreende as seguintes subfases:

1) construção da solução;
2) planejamento do teste;
3) execução manual do teste;
4) codificação da solução;
5) teste com o uso do computador.

Etapa 4 - Avaliação do processo e seus resultados

O trabalho do resolvedor de problemas não pára quando uma solução está pronta. Devemos avaliar a qualidade da solução, o processo que realizamos e questionar as possibilidades de uso posterior da solução obtida e também do próprio método utilizado. Novamente devemos fazer perguntas: Este foi o melhor caminho que poderia ter sido usado? Será que desdobrando a solução não obtenho componentes que poderei usar mais facilmente no futuro? Se esta solução for generalizada é possível reusá-la mais facilmente em outras situações? Registre tudo, organize-se para a resolução de outros problemas. Anote suas decisões, enriqueça a sua biblioteca de soluções e métodos.

3.4. Um pequeno exemplo

Enunciado: Deseja-se escrever um programa que permita determinar a menor quantidade de cédulas necessárias para pagar uma dada quantia em Reais.

Etapa 1 – Compreensão

Questão: Quais os dados de entrada?
Resposta: A quantia a ser paga.

Questão: Qual o resultado a ser obtido?
Resposta: A menor quantidade de cédulas.

Questão: Qual a relação que existe entre a entrada e a saída?
Resposta: O somatório dos valores de cada cédula utilizada deve ser igual à quantia a ser paga.

Questão: Existe algum elemento interno, ou seja, uma dado implícito?
Resposta: Sim, os tipos de cédulas existentes.Considere que o Real possui apenas as seguintes cédulas: 1, 5, 10, 50 e 100. É importante observar que qualquer cédula é um múltiplo de qualquer uma das menores.

Etapa 2 – Planejamento

Conheço algum problema parecido? Existe um problema mais simples?

Podemos entender como um problema mais simples um que busque determinar quantas cédulas de um dado valor são necessárias para pagar a quantia desejada. Por exemplo, para pagar R$ 289,00 poderíamos usar 289 cédulas de R$ 1,00. Ou também poderíamos usar 5 notas de R$ 50,00, mas neste caso ficariam ainda faltando R$ 39,00. Claro que não queremos que falte nem que sobre e, além disso, desejamos que a quantidade seja mínima. Parece uma boa estratégia começar vendo o que dá para pagar com a maior cédula e determinar quando falta pagar. O restante pode ser pago com uma cédula menor, e por aí afora. Explorando a instância do problema já mencionada, vamos fazer uma tabela com estes elementos.

Quantia de dinheiro	Valor da Cédula	Quantidade de cédulas	Resto
289,00	100	289 / 100 = 2	89,00
89,00	50	89 / 50 = 1	39,00
39,00	10	39 / 10 = 3	9,00
9,00	5	9 / 5 = 1	4,00
4,00	1	4 / 1 = 4	0,00
	TOTAL	**11**	

Etapa 3 - Desenvolvimento

Solução 1 - Considerando a tabela acima descrita, definimos a seguir a função **nCedulasA** em que cada linha da coluna "Quantidade de cédulas" corresponde a uma sub-expressão da definição que, somadas, retornam o resultado esperado (note que a operação "//" calcula a divisão inteira de números inteiros, e a operação "%" calcula o resto da divisão inteira entre dois números).

```
>>> def nCedulasA(q):
return q//100 + (q % 100)//50 + ((q % 100) % 50)//10 +
(((q % 100) % 50) % 10)//5 + ((((q % 100) % 50) % 10) % 5)//1
>>> nCedulasA(289)
11
```

Utilizando simplificadamente o esquema proposto no capítulo anterior para avaliar a expressões, temos:

```
>>> nCedulasA(289)
➔ 289//100 + (289 % 100)//50 + ((289 % 100) % 50)//10 + (((289 % 100) %
  50) % 10)//5 + ((((289 % 100) % 50) % 10) % 5)//1
➔ 2 + 89//50 + (89%50)//10 + ((89%50)%10)//5 + (((89%50)%10)%5)//1
➔ 2 + 1 + 39//10 + (39 % 10)//5 + ((39 % 10) % 5)//1
➔ 2 + 1 + 3 + 9//5 + (9 % 5)//1
➔ 2 + 1 + 3 + 1 + 4//1
➔ 2 + 1 + 3 + 1 + 4
➔ 11
```

Solução 2 - Considerando uma propriedade das cédulas, ou seja, já que uma cédula qualquer é múltiplo das menores, a determinação do resto não precisa considerar as cédulas maiores do que a cédula que estamos considerando em um dado ponto.

```
>>> def nCedulasB(q): return q // 100 + (q % 100)//50 + (q %
50)//10 + (q % 10)//5 + (q % 5)//1
>>> nCedulasB(289)
11
```

Etapa 4 – Avaliação do processo

A solução deixa de explicitar as abstrações referentes à quantidade de cédulas de um determinado valor, assim como o resto correspondente. Podemos questionar: não

seria melhor explicitar? Assim, além de podermos usá-las de forma independente, a solução fica mais clara e, portanto, inteligível.

```
>>> def nCedulasC(q): return n100(q) + n50(q) + n10(q) + n5(q) + n1(q)
>>> def n100(q): return q // 100
>>> def n50(q): return resto100(q) // 50
>>> def n10(q): return resto50(q) // 10
>>> def n5(q): return resto10(q) // 5
>>> def n1(q): return resto5(q) // 1
>>> def resto100(q): return q % 100
>>> def resto50(q): return q % 50
>>> def resto10(q): return q % 10
>>> def resto5(q): return q % 5
>>> nCedulasC(289)
11
```

Finalmente, se não quisermos generalizar todas as funções menores ainda, podemos escrever o programa usando definições de constantes locais.

```
>>> def nCedulasD(q):
        def n100(): return q // 100
        def n50(): return resto100() // 50
        def n10(): return resto50() // 10
        def n5(): return resto10() // 5
        def n1(): return resto5() // 1
        def resto100(): return q % 100
        def resto50(): return resto100() % 50
        def resto10(): return resto50() % 10
        def resto5(): return resto10() % 5
        return n100() + n50() + n10() + n5() + n1()
>>> nCedulasD(289)
11
```

Ademais, poderia ser considerada uma possível mudança no sistema, de modo a incluir uma nova cédula que não seja múltiplo de seus valores menores.

3.5. Provérbios

O professor Polya também nos sugere que a lembrança de alguns provérbios pode ajudar o aprendiz (e o resolvedor de problemas) a organizar o seu trabalho. Diz Polya que, apesar dos provérbios não se constituírem em fonte de sabedoria universalmente aplicável, seria uma pena desprezar a descrição pitoresca dos métodos heurísticos que apresentam.

Alguns são de ordem geral, tais como:

O fim indica os meios.

Seus melhores amigos são O que, Por que, Onde, Quando e Como. pergunte O que, pergunte Por que, pergunte Onde, pergunte Quando e pergunte Como - e não pergunte a ninguém quando precisar de conselho.

Não confie em coisa alguma, mas só duvide daquilo que merecer dúvida.

Olhe em torno quando encontrar o primeiro cogumelo ou fizer a primeira descoberta; ambos surgem em grupos.

A seguir apresentamos uma lista deles, organizados pelas etapas às quais parecem mais relacionados.

Etapa 1 : Compreensão do problema.

Quem entende mal, mal responde.

Pense no fim antes de começar.

O tolo olha para o começo, o sábio vê o fim.

O sábio começa pelo fim, o tolo termina no começo.

Etapa 2 : Planejamento da solução.

A perseverança é a mãe da boa sorte.

Não se derruba um carvalho com uma só machadada.

Se no princípio não conseguir, continue tentando.

Experimente todas as chaves do molho.

Veleja-se conforme o vento.

Façamos como pudermos se não pudermos fazer como queremos.

O sábio muda de opinião, o tolo nunca.

Mantenha duas cordas para um arco.

Faça e refaça que o dia é bastante longo.

O objetivo da pescaria não é lançar o anzol mas sim pegar o peixe.

O sábio cria mais oportunidades do que as encontra.

O sábio faz ferramentas daquilo que lhe cai às mãos.

Fique sempre de olho na grande ocasião.

Etapa 3: Construção da solução.

Olhe antes de saltar.

Prove antes de confiar.

Uma demora prudente torna o caminho seguro.

Quem quiser navegar sem risco, não se faça ao mar.

Faça o que puder e espere pelo melhor.

É fácil acreditar naquilo que se deseja.

Degrau a degrau sobe-se a escada.

O que o tolo faz no fim, o sábio faz no princípio.

Etapa 4: Avaliação da solução.

Não pensa bem quem não repensa.

É mais seguro ancorar com dois ferros.

EXERCÍCIOS

1. Compare as definições para a função que descreve o número mínimo de cédulas para pagar uma dada quantia e discuta;
2. Desenvolva a solução para o problema das cédulas considerando o fato de que o Real possui notas de 2 e notas de 20. O que muda?
3. Apresente três problemas semelhantes ao das cédulas;
4. Desenvolva uma solução para o problema considerando que para cada tipo de cédula existe uma quantidade limitada (maior ou igual a zero);

4. ABSTRAÇÃO, GENERALIZAÇÃO, INSTANCIAÇÃO E MODULARIZAÇÃO

Na busca por resolver um problema podemos usar vários princípios, cada um evidentemente terá uma utilidade para a resolução do problema, ou para garantia de sua correção ou ainda para facilitar os usos posteriores da solução que obtivermos. Apresentamos a seguir alguns deles.

4.1. Abstração

Quando escrevemos uma expressão e não damos nome a ela, o seu uso fica limitado àquele instante específico. Por exemplo, suponha que desejamos determinar a hipotenusa de um triângulo retângulo com catetos 10 e 4. Como conhecemos o teorema de Pitágoras ($a^2 = b^2 + c^2$), podemos usar diretamente nossa máquina funcional para avaliar a seguinte expressão:

```
>>> from math import sqrt
>>> sqrt((10 * 10) + (4 * 4))
10.770329614269007
```

A expressão que codificamos serve apenas para esta vez. Se em algum outro instante precisarmos avaliá-la, teremos que codificá-la novamente.

Para evitar isso, é que damos nomes às nossas expressões, para que possamos usá-las repetidamente, apenas referenciando-as pelo seu nome. No caso acima, poderíamos escrever a definição:

```
>>> def hipotenusa(): return sqrt((10 * 10) + (4 * 4))
```

De posse dessa definição nossa máquina poderá avaliar a expressão sempre que dela precisarmos. Basta escrevê-la:

```
>>> hipotenusa()
10.770329614269007
```

Você pode agora estar se perguntando, porque não trocamos a definição para usar diretamente o valor **`10.7703...`**?

```
>>> def hipotenusa(): return 10.770329614269007
```

Agindo assim a máquina não precisaria avaliar a expressão **`sqrt((10 * 10) + (4 * 4))`** a cada uso. Por outro lado não ficaria registrada a origem do valor **`10.7703...`**, com o tempo perderíamos esta informação. De qualquer forma, teríamos criado uma **abstração** à qual denominamos **`hipotenusa`**.

Outra pergunta pode estar rondando a sua cabeça, por que escrever uma definição que é sempre avaliada para o mesmo valor, por que não generalizá-la? Bom, este foi apenas um recurso didático para separar os dois conceitos: a abstração que acabamos de apresentar e a generalização que apresentaremos a seguir. No entanto convém lembrar que algumas definições são realmente constantes e que são de grande utilidade, como é o caso da definição de **`pi`**, como vimos anteriormente.

4.2. Generalização

Quando uma mesma abstração se aplica a vários casos podemos generalizá-la. Assim, podemos usá-la várias vezes para os mesmos valores ou ainda para valores diferentes. Porquanto, dizemos que a **generalização** promove o reuso da definição.

Uma forma comum de generalizar é através do conceito de parametrização. Esta consiste em indicar na interface da função quais são os objetos da generalização. Para o exemplo que estamos trabalhando, podemos escrever:

```
>>> def hipotenusa(x, y): return sqrt((x * x) + (y * y))
```

Temos então descrita a função paramétrica **`hipotenusa`** que generaliza tal abstração para diferentes valores de catetos, representados pelos parâmetros **`x`** e **`y`**.

4.3 Instanciação

A parametrização permite que usemos a mesma definição para diferentes instâncias do problema. Por exemplo, suponha que desejemos determinar a hipotenusa de três triângulos retângulos. O primeiro com catetos 10 e 4, o segundo com catetos 35 e 18 e o terceiro com catetos 9 e 12. Assim, podemos estabelecer a **instanciação** do problema do cálculo da hipotenusa para os triângulos citados informando os catetos como parâmetros da função, conforme segue:

```
>>> hipotenusa(10, 4)
10.770329614269007
>>> hipotenusa(35, 18)
39.357337308308857
>>> hipotenusa(9, 12)
15.0
```

4.4. Modularização

Em geral, nossos problemas não serão tão simples e diretos quanto o exemplo acima. Quando nos deparamos com problemas maiores, um bom princípio é:

Dividir para facilitar a conquista.

Basicamente este princípio consiste em:

i) quebrar o problema inicial em problemas menores;

ii) elaborar a solução para cada um dos problemas menores e;

iii) combinar estes subproblemas para obter a solução do problema inicial.

A cada um dos subproblemas encontrados podemos reaplicar o mesmo princípio. Segundo Polya, esta é uma heurística muito importante à qual ele denomina de **decomposição** e **combinação**. Antes de pensar em codificar a solução em uma linguagem de programação específica, podemos representá-la através de um esquema gráfico denominado de estrutura modular do problema (veja a representação para o exemplo a seguir).

MODULARIZANDO: Considere o problema de descrever a área da Figura 4.1 abaixo.

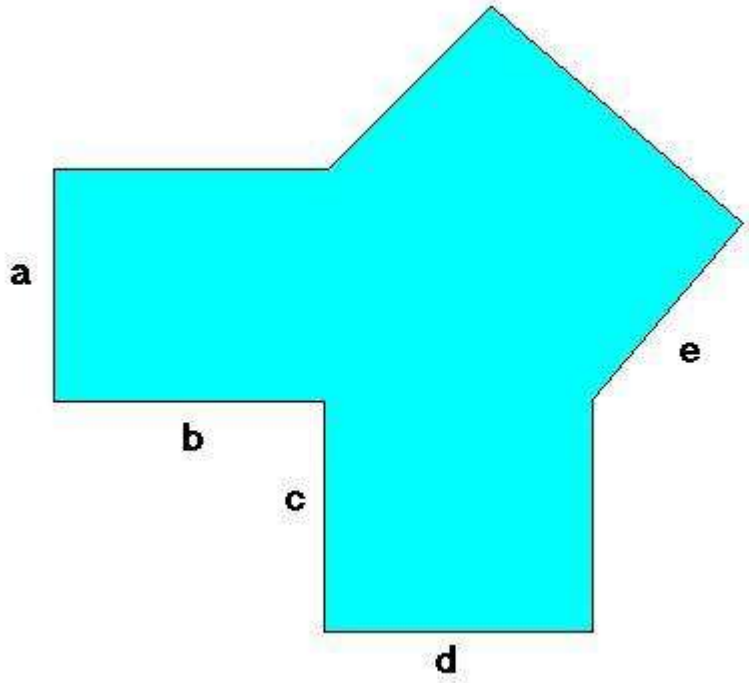
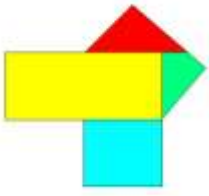


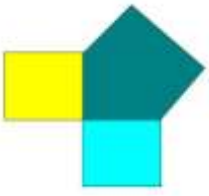
Figura 4.1 Um polígono irregular para o qual desejamos determinar a sua área

Como podemos concluir por uma inspeção da figura, não existe uma fórmula pronta para calcular sua área. Precisamos dividir a figura em partes para as quais conheçamos uma maneira de calcular a área, como triângulos e retângulos, e que, além disso, conheçamos as dimensões necessárias.

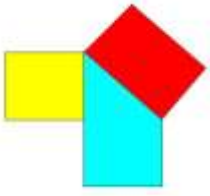
Podemos fazer várias tentativas, como por exemplo, as ilustradas nas Figuras 4.2 a, 4.2 b e 4.2 c.



a



b



c

Figura 4.2 – Possíveis subdivisões do polígono apresentado na Figura 4.1.

Podemos tentar descrever a área das figuras menores em cada uma das figuras apresentadas. Nas Figuras 4.2.b e 4.2c, parece que precisaremos subdividir novamente. Em 4.2 b a “casinha” pode ser transformada em um retângulo e um triângulo. Em 4.2 c é a “casinha” azul (meia-água) que pode ser dividida em um retângulo e um triângulo, como na Figura 4.3. E a Figura 4 a? Que podemos dizer?

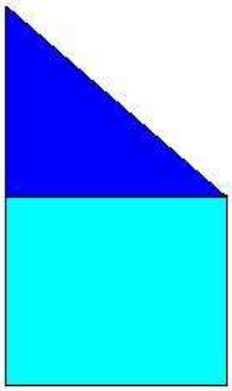


Figura 4.3 – Subdivisão de uma sub-figura da Figura 4.3 c,

Vamos partir para a nossa solução a partir da figura Fig. 4.2c. Podemos dizer que a área total pode ser obtida pela soma das áreas amarela, vermelha e azul. A área azul pode ser subdividida em azul-claro e azul-escuro (ver Figura 4.3).

área total = área amarela + área vermelha + área azul
área azul = área azulclaro + área azulescuro

Quando escolhemos as áreas acima citadas, não foi por acaso. A escolha foi baseada na simplicidade do subproblema. Podemos usar este conhecimento para chegar a um outro. Três das áreas que precisamos calcular são de forma retangular. Ou seja, são especializações de um conceito mais geral. A quarta área, também pode ser obtida pela especialização do conceito de triângulo retângulo.

Vamos agora aproximar nossas definições da linguagem de programação.

Portanto podemos escrever:

```
>>> def areaRetangulo(x, y): return x * y
```

o triângulo que temos é retângulo e podemos descrever sua área por:

```
>>> def areaTrinaguloRetangulo(x, y): return (x * y)/2
```

A determinação da área vermelha nos traz uma pequena dificuldade. Não nos foi informado qual o comprimento da base do retângulo. E agora? Observando bem a figura podemos concluir que a base do retângulo é igual à hipotenusa do triângulo retângulo de catetos ***a*** e ***d.*** Podemos escrever então:

```
>>> def areaTotal(a, b, c, d, e): return areaRetangulo(a, b) +
areaRetangulo(hipotenusa(a, d), e) + areaAzul(a, c, d)
>>> def areaAzul(a, c, d): return areaRetangulo(c, d) +
areaTrianguloRetangulo(a, d)
```

A estrutura da solução de um problema obtida pela modularização pode ser representada por um diagrama denominado árvore. Para o problema acima discutido a árvore é da forma apresentada na Figura 4.5.

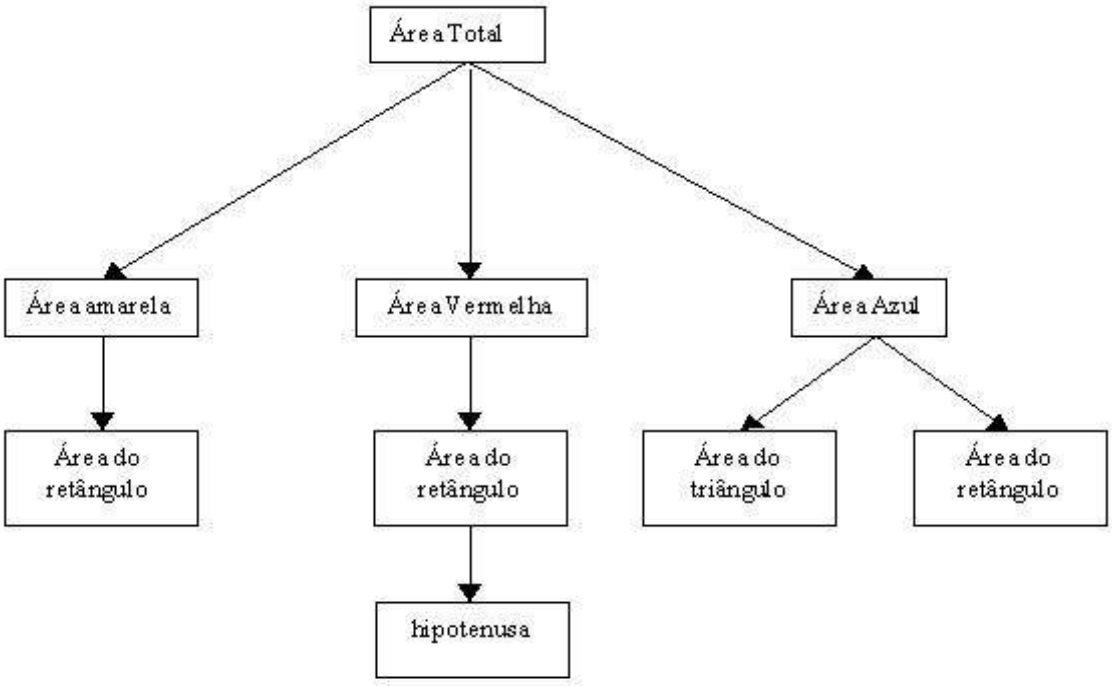


Figura 4.5 – Representação em árvore da estrutura modular do problema "cálculo da área do polígono irregular apresentado na Figura 4.1".

4.5. Um exemplo detalhado

Problema: Escreva uma descrição funcional para o volume de cada uma das peças apresentadas nas Figuras 4.6 e 4.7.

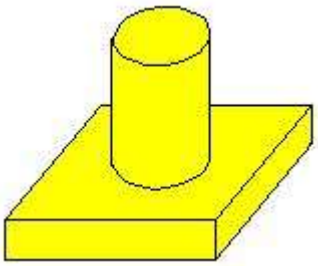


Figura 4.6 – Peça no. 1

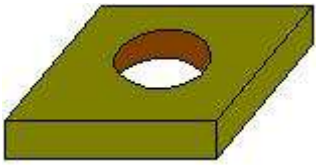


Figura 4.7 – Peça no. 2

Solução 1: Vamos começar pela Figura 4.6. Uma rápida análise nos leva a identificar duas partes. Na parte inferior temos um paralelepípedo, sobre o qual se assenta um cilindro. Vamos supor que são maciços. Chamemos de *a*, *b* e *c* as dimensões do paralelepípedo, de *r* o raio da base do cilindro e de *h* a sua altura. O volume total da peça pode ser determinado pela soma do volume das duas peças menores. Chamemos a função de volfig46. Uma possível definição para essa função é:

```
>>> def volumePeca1(a, b, c, r, h): return (a * b * c) + (pi() * r * r * h)
```

Vamos então tratar da peça descrita na figura 4.7. Agora identificamos uma peça apenas. Um paralelepípedo com um furo no centro. Chamemos de *a*, *b* e *c* as dimensões do

paralelepípedo, de *r* o raio do buraco. Para descrever o volume da peça devemos subtrair do volume do paralelepípedo o volume correspondente ao buraco. Chamemos a função de volfig47. Uma possível definição para volfig47 é:

```
>>> def volumePeca2(a, b, c, r): return (a * b * c) + (pi() * r * r * c)
```

Solução 2: A solução 1, apesar de resolver o problema, deixa de contemplar algumas práticas importantes. No tratamento da primeira peça (Figura 4.6), apesar de identificadas duas peças menores, estas abstrações não foram descritas separadamente. Ou seja, deixamos de registrar formalmente a existência de duas abstrações e com isso não pudemos modularizar a descrição da função principal. Podemos tentar então um outro caminho, contemplando as abstrações para o cilindro e para o paralelepípedo:

```
>>> def volumeCilindro(r, h): return pi() * r * r * h
>>> def volumeParalelepipedo(a, b, c): return a * b * c
>>> def volumePeca1(a, b, c, r, h): return volumeParalelepipedo(a, b, c) +
    volumeCilindro(r, h)
```

Voltemos então para a segunda peça (Figura 4.7), e vejamos se podemos identificar similaridades. Aqui só temos uma peça, que se parece com o paralelepípedo da primeira figura. Como podemos identificar similaridades? Aprofundando melhor nossa análise podemos lembrar que o furo no paralelepípedo, corresponde a um cilindro que foi retirado. Desta forma, podemos então concluir que o volume da figura 4.7 pode ser obtido pela diferença entre o volume de um paralelepípedo e de um cilindro. Como já temos as definições para volume de cilindro e volume de paralelepípedo, só nos resta escrever a definição final do volume da figura 4.7:

```
>>> def volumePeca2(a, b, c, r): return volumeParalelepipedo(a, b, c) -
    volumeCilindro(r, c)
```

Analisando a solução, podemos tirar algumas conclusões:

a) a solução ficou mais clara,
b) a solução propiciou o reaproveitamento de definições.
c) se precisarmos usar volume de cilindro e de paralelepípedo isoladamente ou em outras combinações, já as temos disponíveis.

Considerações complementares - As descrições das áreas de um retângulo e de uma circunferência podem ser dadas respectivamente por:

```
>>> def areaCirculo(r): return pi() * r * r
>>> def areaRetangulo(a, b): return a * b
```

Desta forma, podemos reescrever o volume do cilindro e do paralelepípedo da seguinte maneira:

```
>>> def volumeCilindro(r, h): return areaCirculo(r) * h
>>> def volumeParalelepipedo(a, b, c): return areaRetangulo(a, b) * c
```

Solução 3: Se aprofundarmos a análise podemos observar que as duas figuras podem ser abstraídas como uma só! O cilindro que aparece na primeira, como um volume que deve ser acrescentado pode ser entendido como o mesmo cilindro que deve ser subtraído na segunda. Além disso, podemos estender a solução para furos que não vazem a peça, pois a altura do cilindro pode ser qualquer. Considere a definição a seguir:

```
>>> def volumePeca(a, b, c, r, h): return volumeParalelepipedo(a, b, c) +
    volumeCilindro(r, h)
```

Podemos observar que esta é a mesma definição apresentada anteriormente para a Figura 4.6. O que muda é o uso. Para calcular o volume da Figura 4.6 usamos um ***h*** positivo e para Figura 4.7 um ***h*** negativo. Observe os exemplos a seguir, considerando um paralelepípedo de medidas a = 6, b = 8, c = 2:

`>>> volumePeca(6, 8, 2, 2, 5)` `158.83199999999999`	-- determina o volume de uma instância da figura Fig. 4.6, com um cilindro de raio 2 e altura 5.
`>>> volumePeca(6, 8, 2, 2, -2)` `70.867199999999997`	-- determina o volume de uma instância da figura Fig. 4.7, vasada por um furo de raio 2. -- neste caso fornecemos um valor negativo para a altura do cilindro com o mesmo valor da altura do paralelepípedo
`>>> volumePeca(6, 8, 2, 2, -1)` `83.433599999999998`	-- determina o volume de uma instância da figura Fig. 4.7, que tem um furo de raio 2 e profundidade 1.

EXERCÍCIOS:

1. Resolva o problema da Figura 4.1 u**s**ando a modularização sugerida pela figura 4.2 b.
2. Redesenhe a Figura 4.5 (árvore de modularização) para a solução apresentada ao exercício 2.
3. Apresente 3 problemas similares para cálculo de área de polígono irregular.
4. Apresente 3 problemas similares para cálculo de volume de objetos.
5. Apresente 3 problemas similares em outros domínios de conhecimento.

5. TIPOS DE DADOS BÁSICOS: NUMÉRICO E CARACTER

5.1. Introdução

Denominamos **Tipo de Dados** a um conjunto de valores, munido de um conjunto de operações sobre esses valores. Por exemplo, podemos denominar de T1 ao tipo de dados formado por um conjunto S de valores idêntico aos números naturais (S = {0,1,2,3, ...}) e munido das operações de adição (***a***) e multiplicação (***m***).

Cada operação, por sua vez, relaciona um (ou mais) conjunto(s) de valores do domínio a outro conjunto de valores do contradomínio. Para o tipo T1, o domínio de ***a*** é o produto cartesiano ***S X S*** e o contradomínio é ***S.*** A notação a seguir, semelhante à notação matemática, é usualmente utilizada e em geral é denominada de "assinatura" da operação.

a :: S X S → S
m :: S X S → S

Esta notação significa que a operação de adição **a** opera dois valores do conunto S e resulta em um valor do mesmo conjunto. Por exemplo, a(3,4) resulta em 7.

Os tipos de dados básicos da linguagem Python que estudaremos neste capítulo são os tipos numéricos, que usamos até agora seguindo a formação matemática que já temos, e o tipo caracter. Este último é brevemente apresentado neste capítulo, pois as operações sobre ele dependem de conceitos que são apresentados em capítulos mais avançados, quanto então poderemos estudar este tipo mais detalhadamente. Outro tipo básico, chamado boolean, é apresentado no capítulo seguinte a este.

5.2. Tipos Numéricos

5.2.1 Números Inteiros:

Para trabalhar com números inteiros, a linguagem Python provê os tipos **Int** e **Long**. O primeiro representa números inteiros de precisão fixa, enquanto o segundo representa números inteiros de precisão arbitrária, capaz, em princípio, de produzir números com uma quantidade ilimitada de algarismos. Entretanto, como a memória do computador é finita, qualquer que seja a máquina real que estivermos usando, inevitavelmente esbarrará em limites. Na ilustração a seguir podemos observar essa flexibilidade, quando obtemos o valor para a expressão 2^{1000}, um número de 302 algarismos. Claro, o limite pode estar bem longe e podemos não atingí-lo em nossas aplicações.

```
>>> 2**1000
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
```

Experimente com números maiores! Por exemplo, 9999^{9999} é um número que tem por volta de 40 mil algarismos.

5.2.3 Números Reais:

Para trabalhar com números reais de precisão variável, a linguagem Python provê o tipo **Float**. Também neste caso, a capacidade do computador delimita a precisão e magnitude que se pode alcançar. Como na notação científica, a precisão nos diz quantos algarismos significativos são usados e a magnitude nos diz qual o maior expoente admitido. É importante observar que, nesta linguagem, as casas decimais são determinadas por "." (ponto) e não por "," (vírgula).

```
1.05 , 10.5 , -100.0005
```

Uma outra notação possível para números reais é a notação científica, denotada por uma base, seguida da letra 'e' ou 'E' e o expoente.

```
>>> 10e-5
0.0001
```

5.2.3 Operadores Numéricos:

A tabela a seguir apresenta algumas operações primitivas fornecidas pela linguagem para manipulação dos tipos numéricos:

Nome	Descrição	Exemplos	
+	Adição	>>> 1 + 5 + 20 + 31 57	>>> 1.0 + 5.0 6.0
-	Subtração	>>> 1234 - 4321 -3087	>>> 1234.0 - 4321.0 -3087.0
*****	Multiplicação	>>> 20 * 10 * 98 19600	>>> 20.0 * 10.0 * 98.0 19600.0
/	Divisão Real	>>> 30/20 1.5	>>> 30.0/20.0 1.5
//	Divisão inteira	>>> 30//20 1	>>> 30.0//20.0 1.0
%	Resto da divisão inteira	>>> 30%20 10	>>> 30.0%20.0 10.0
******	Potência	>>> 2**20 1048576	>>> 2.0**20.0 1048576.0
abs	Valor absoluto	>>> abs(-123) 123	>>> abs(-123.0) -123.0

As operações acima descritas podem ainda ser combinadas para construir expressões mais complexas:

```
>>> 2 ** 2 + 12 * 3
40
```

Contudo, há que se observar que, assim como acontece na matemática, as operações seguem uma ordem de precendência. No exemplo dado, realiza-se primeito a potência, seguido da multiplicação e por último a soma.

De forma geral, as operações obedecem à seguinte ordem de precendência:

** >> % >> // >> / >> * >> - >> +

Tal precedência pode modificada com o uso de parênteses, de forma que as operações delimitadas por parênteses têm precedência sobre aquelas definidas fora dos parênteses. Vejamos uma versão adaptada do exemplo anterior com uso de parênteses para modificar a precendência:

```
>>> 2 ** (2 + 12) * 3
49152
```

Observa-se que, conforme explicitado no esquema de avaliação a seguir, o uso de parênteses no segundo caso deu à operação de soma precedência sobre as outras.

2 ** 2 + 12 * 3 ➔ 4 + 12 * 3 ➔ 4 + 36 ➔ 40

2 ** (2 + 12) * 3 ➔ 2 ** 24 * 3 ➔ 16384 * 3 ➔ 49152

5.2.3 Conversão Numérica:

Quando uma das operações é realizada entre diferentes tipos numéricos, o Python faz a coerção (ou conversão) automaticamente para o tipo mais abrangente. Por exemplo:

```
>>> 1 + 4.5
5.5
```

é equivalente a

```
>>> 1.0 + 4.5
5.5
```

Ou seja, o Python faz automaticamente a coerção do valor inteiro 1 para o valor real 1.0 para que a operação de soma seja realizada corretamente.

Ademais, pode-se fazer a coerção forçada usando as seguintes funções:

Nome	Descrição	Exemplos
float(x)	Converte um número inteiro em um número real	`>>> float(3)` `3.0`
int(x)	Converte um número real em um número inteiro, eliminando a parte decimal	`>>> int(3.75)` `3`

5.2.3 Outras Operações:

Nome	Descrição	Exemplos
round(x)	Arredonda um número real	`>>> round(3.75)` `4.0`

5.3. Tipo Caracter

Este tipo representa letras, símbolos e até mesmo números (como símbolos) e é representado entre aspas simples ou duplas. A um conjunto de caracteres é dado o nome de *string*. Por exemplo

Caracteres: ‘a’ ‘b’ ‘5’ ‘@’ ‘.’
String: ‘ab5@.’

Assim como os tipos numéricos, este tipo também pode ser usado na definição de funções:

```
>>> def Ola():
    return "Ola Mundo!!!"
>>> Ola()
'Ola Mundo!!'
```

6. EXPRESSÕES LÓGICAS E O TIPO DE DADOS *BOOLEAN*

6.1. Introdução

Uma característica fundamental dos agentes racionais é a capacidade de tomar decisões adequadas considerando as condições apresentadas pelo contexto onde está imerso. Uma máquina que sabe apenas fazer contas, ou seja, manusear as operações aritméticas, terá sua utilidade fortemente reduzida e, portanto não despertará tantos interesses práticos. Os computadores que temos hoje são passíveis de serem programados para tomada de decisão, ou seja, é possível escolher entre duas ou mais ações, aquela que se deseja aplicar em um determinado instante. Em nosso paradigma de programação precisamos de dois elementos fundamentais: um tipo de dados para representar a satisfação ou não de uma condição e um mecanismo que use essas condições na escolha de uma definição. Neste capítulo discutiremos a natureza das proposições lógicas, sua aplicabilidade na resolução de problemas e introduziremos um novo tipo de dados, denominado **boolean**. Satisfazendo logo de saída a curiosidade do leitor lembramos que o nome é uma homenagem a George Boole que estudou e formalizou as operações com estes tipos de valores.

6.2. Proposições Lógicas

Revirando o baú das coisas estudadas no ensino fundamental, por certo encontraremos as sentenças matemáticas. Lembraremos então que elas são afirmações sobre elementos matemáticos, tais como os exemplos a seguir:

O conceito de proposição lógica é mais geral, aplicando-se às mais diversas situações do cotidiano, como nos exemplo a seguir:

1. Maria é namorada de Pedro
2. José é apaixonado por Maria
3. Hoje é domingo

Analisando o significado da informação contida em uma proposição lógica, podemos concluir que ela será uma afirmação verídica quando se referir a fatos que realmente acontecem em um determinado mundo. Quando isso não ocorre, concluímos que elas são falsas. Para podermos avaliar uma dada proposição, é necessário ter acesso ao mundo considerado.

Sentenças Fechadas: as sentenças 1 a 6 possuem uma característica importante: todos os seus componentes estão devidamente explicitados. Denominamos estas sentenças de sentenças fechadas. Uma sentença fechada pode ser avaliada imediatamente, conferindo o que elas afirmam com o mundo sobre o qual elas se referem.

Sentenças Abertas: são outro tipo de proposição importante. Nestas, alguns personagens não estão devidamente explicitados e, portanto a sentença não pode ser avaliada. Quando tratamos sentenças abertas, antes é necessário instanciá-las para algum valor. Por exemplo, a sentença matemática

$$x + 5 > 10$$

nem sempre é verdadeira. Depende do valor que atribuirmos à variável ***x***. Quando atribuímos valores às variáveis de uma sentença dizemos que estamos instanciando a sentença.

No caso acima, podemos instanciar a sentença para os valores 3 e 10, entre outros, obtendo as seguintes instâncias:

$$3 + 5 > 10$$
$$10 + 5 > 10$$

Agora podemos avaliá-las e concluir que a segunda é verídica e a primeira não.

Sentenças Compostas: O discurso do cotidiano e até o discurso matemático podem ser escritos como uma lista de proposições simples. Exemplos:

1. Hoje é domingo
2. Aos domingos tem futebol
3. Quando meu time joga, eu assisto

Nem sempre isto é desejável ou suficiente. Para resolver a questão, podemos contar com as sentenças compostas. Como por exemplo:

a) *Três é menor que cinco e o quatro também;*
b) *Domingo irei ao futebol ou escreverei notas de aula;*
c) *Esperanto não é uma linguagem de programação.*

Observamos então, o surgimento de três palavras para criar essas sentenças e que essas palavras (***e***, ***ou***, ***não***) não se referem a coisas, propriedades ou fenômenos de um determinado universo. Elas são denominadas de palavras lógicas, visto que a sua função na linguagem é possibilitar a articulação entre as proposições do discurso.

A composição de uma proposição pode envolver várias palavras lógicas, como nos exemplos a seguir:

1. ***Dois é menor que três e maior que um mas não é maior que a soma de três com dois;***
2. ***Maria gosta de Pedro e de Joana mas não gosta de Antonio.***

Avaliação de Sentenças Compostas: Para avaliar sentenças simples, vimos que bastava inspecionar o mundo ao qual ela se refere e verificar se a situação expressa pela sentença ocorre ou não. E como proceder para as sentenças compostas? Um caminho que parece confiável é apoiar essa avaliação no papel representado por cada uma das palavras lógicas. Assim, ao considerarmos a proposição

Hoje fui ao cinema e ao teatro,

só poderemos dizer que ela é verídica se tanto a proposição ***Hoje fui ao cinema*** quanto a proposição ***Hoje fui ao teatro*** forem avaliadas como verídicas.

Para a proposição composta

Maria foi à missa ou ao cinema,

devemos considerá-la como verídica se uma das duas proposições:

1. ***Maria foi à missa***
2. ***Maria foi ao cinema***

forem avaliadas como verídica. E se as duas forem avaliadas como verídicas? No discurso cotidiano tendemos a pensar nesta situação como inverídica visto que queríamos uma ou outra. A linguagem natural não estabelece se devemos ou não explicitar que não estamos falando do caso em que ambas podem ser constatadas. Podemos assumir, portanto, que a nossa sentença composta usando ***ou*** será avaliada como verídica quando pelo menos uma das duas proposições que a compõe for avaliada com verídica.

No caso da sentença composta usando a palavra lógica ***não***, como em

Hoje não choveu

diremos que ela será avaliada como verídica quando a proposição ***Hoje choveu*** não puder ser constatada e como inverídica no caso oposto.

6.3. O Tipo de Dados Boolean

Podemos agora discutir sobre a natureza do valor resultante da avaliação de uma sentença. A avaliação é de natureza funcional, ou seja, dada uma sentença ***s*** ela será mapeada em um de dois valores distintos. Então, o contradomínio de uma avaliação é um conjunto com apenas dois valores, um para associar com avaliações verídicas e outras com as inverídicas. Podemos escolher um nome para esses valores. Historicamente, as linguagens de programação os tem denominado de ***True*** e ***False***. O primeiro para associar com as proposições que podem ser constatadas no mundo considerado e a segunda com as não constatáveis. Para a avaliação das proposições compostas, podemos considerar uma função matemática cujo domínio e contradomínio são definidos como o conjunto que acabamos de definir com as constantes ***True*** e ***False***. Assim

aval :: <sentença> → {True, False}

As proposições compostas podem ser formadas pelas operações lógicas de conjunção, disjunção ou negação. Em Haskell, essas operações são representadas por:

Operação lógica	Operador lógico (Python)
e	and
ou	or
não	not

Vamos então formalizar a avaliação de sentenças compostas. Sejam s1 e s2 duas proposições lógicas:

1. O valor lógico da sentença ***s1* and *s2*** é ***True*** se e somente se o valor lógico de ***s1*** é ***True*** e o valor lógico de ***s2*** é ***True*** e é ***False*** em caso contrário;
2. O valor lógico da sentença ***s1* or *s2*** é ***False*** se e somente se o valor lógico de ***s1*** é ***False*** e o valor lógico de ***s2*** é ***False*** e é ***True*** em caso contrário:
3. O valor lógico da sentença ***not s1*** é ***True*** se e somente se o valor lógico de ***s1*** é ***False*** e é ***False*** em caso contrário.

Tabela Verdade: Estas definições também podem ser representadas através de uma enumeração de suas associações, formando o que se costuma chamar de ***Tabelas Verdade*** as quais apresentamos a seguir.

s1	s2	s1 and s2
True	True	True
True	False	False
False	True	False
False	False	False

s1	s2	s1 or s2
True	True	True
True	False	True
False	True	True
False	False	False

s1	not s1
True	False
False	True

6.4. Operadores Relacionais

Quando falamos de proposições lógicas no discurso matemático, ou melhor, em sentenças matemáticas, usamos termos tais como: “menor do que”, “menor ou igual”, “diferente”, entre outros. Estes termos são fundamentais para a nossa intenção de prover os computadores com a capacidade de decisão. Denominamos estes elementos de **operadores relacionais**, pois estabelecem uma relação de comparação entre valores de um mesmo domínio. O contradomínio deles é do tipo Boolean. A tabela a seguir apresenta esses operadores, seus significados, suas representações em Python e exemplos de uso.

Operador	*Significado*	*Exemplo*	*Resultado*
==	igualdade	(2 + 3) == (8 – 3)	True
!=	Diferença	5 != (4 * 2 -3)	False
<	Menor	(2 + 3) < 6	True
<=	Menor ou igual	(2 * 3) <= 6	True
>	Maior	(4 + 2) > (2 * 3)	False
>=	Maior ou igual	(8 – 3 * 2) >= 15	False

Podemos usar estes operadores para construir novas definições ou simplesmente, em uma sessão de IDLE na avaliação de uma expressão, como apresentado nos exemplos a seguir.

```
>>> 5 > 4
True
>>> 4 != 4 + 1
True
>>> 5 % 2 == 1
True
>>> 3 > 2
True
>>> 3 < 2
False
```

6.5. Expressões e definições

Agora que temos um novo tipo de dados, podemos utilizá-lo a nosso serviço, escrevendo expressões de forma tão natural quanto aquela que usamos para escrever expressões aritméticas. Usando essas expressões podemos então construir definições cujo tipo resultante seja booleano. Os ingredientes básicos para construir essas expressões são os operadores relacionais.

Expressões simples: Por exemplo, para construir uma definição que avalie se um número é par, podemos usar a seguinte definição:

```
>>> def par(x): return (x % 2) == 0
```

Vejamos alguns exemplos de avaliação da função par:

```
>>> par(2)
True
>>> par(3)
False
```

Outros exemplos de definições:

Verificar se ***a*** é múltiplo de ***b***	`def multiplo(a, b): return (a % b) == 0`
Verificar se ***a*** é divisor de ***b***	`def divisor(a, b): return multiplo(b, a)`
Verificar se uma distância ***d*** é igual à diagonal de um quadrado de lado ***a***	`def diag(d, a): return (a * sqrt(2)) == d`
Verificar se um número é um quadrado perfeito	`def quadp(n): return sqrt(n)**2 == n`
Verificar se dois números ***a*** e ***b*** são termos consecutivos de uma P.A. de razão ***r***	`def spa(a, b, r): return (a + r) == b`

Expressões compostas: Podemos usar agora os operadores lógicos para construir expressões compostas. Veja os exemplos a seguir:

Verificar se 3 números estão em ordem crescente	`def ordc(a, b, c): return (a < b) and (b < c)`

Verificar se um número ***x*** está no intervalo fechado definido por ***a*** e ***b***	`def pert(x,a,b): return (x >= a) and (x <= b)`
	ou
	`def pert(x,a,b): return not((x < a)or(x > b))`

Verificar se um determinado ponto do espaço cartesiano está no primeiro quadrante	`def pquad(x, y): return (x > 0) and (y > 0)`
Verificar se 3 números ***a***, ***b*** e ***c***, são lados de um triângulo retângulo	`def tret(a, b, c): return ((a**2 + b**2) == c**2) or ((a**2 + c**2) == b**2) or ((b**2 + c**2) == a**2)`

Quando nossas expressões possuem mais de um operador lógico devemos observar a precedência de um sobre o outro. Por exemplo, na expressão

P or Q and R

as letras P, Q e R são expressões lógicas. O operador **and** será avaliado primeiro pois tem precedência sobre o **or**, portanto a expressão será avaliada para *True* se P for avaliado para *True* ou se a subexpressão (Q **and** R) for avaliada para *True*.

Podemos modificar esta ordem utilizando parêntesis como nas expressões aritméticas. A expressão acima poderia ser reescrita como

```
(P or Q) and R
```

Agora, para que ela seja avaliada para verdadeira é preciso que R seja avaliada como verdadeira.

Vejamos mais alguns exemplos de definição de função booleana:

Verificar se x está fora do intervalo definido por a e b e a e b estão em ordem não decrescente	`def f(x, a, b): return ((x <= a) or (x >= b)) and (a <= b)`
Verificar se x é menor que a ou se x é maior que b e a e b estão em ordem não decrescente	`def g(x, a, b): return (x <=a) or (x >= b) and (a <= b)`

6.6. Resolvendo um Problema:

Desejamos verificar se um determinado ponto do espaço cartesiano está dentro ou fora de um retângulo paralelo aos eixos, conhecidos o ponto, o canto superior esquerdo e o canto inferior direito do retângulo.

Etapa 1 [Entendendo o problema] O ponto pode estar em qualquer quadrante? E o retângulo, pode estar em qualquer quadrante? Basta ter os dois cantos para definir o retângulo? Em que ordem serão informados os cantos? Como se descreve um canto? Um ponto que esteja sobre um dos lados, está dentro ou fora do retângulo?

Vamos assumir então as seguintes decisões: discutiremos inicialmente apenas sobre pontos e retângulos localizados no primeiro quadrante. A ordem em que os dados serão informados será: primeiro o ponto, depois o canto superior esquerdo e por último o canto inferior direito. Para cada ponto serão informadas as duas coordenadas, primeiro a abscissa e depois a ordenada. Os pontos na borda são considerados pertencentes ao retângulo.

Etapa 2 [Planejando a Solução]: Conheço algum problema parecido? Posso decompor este problema em problemas mais simples? Sei resolver um problema mais geral em que este é um caso particular?

Bom, já nos envolvemos com o problema para verificar se um ponto estava num intervalo linear, este também se refere a intervalo. Verificar se um ponto pertence a uma região qualquer do espaço n-dimensional é mais geral, se eu conhecesse uma definição para este problema geral, bastaria instanciá-la. Será que posso decompor o problema na verificação de dois espaços lineares, um definido pelos lados paralelos ao eixo das ordenadas e outro paralelo ao eixo das abscissas? Se afirmativo, como combino as duas soluções?

Para que um ponto esteja dentro do retângulo é necessário que sua ordenada esteja entre as ordenadas dos dois cantos. Sabemos também que a abscissa precisa estar entre as abscissas dos dois cantos. Isso basta? Para combinar as soluções percebemos que é suficiente que as duas primeiras estejam satisfeitas.

Etapa 3 [Construindo a Solução] Construindo a solução - Anteriormente já elaboramos a definição de pertinência a um intervalo linear, vamos usá-la aqui para definir a pertinência linear e no plano.

```
>>> def pertLinear(x,a,b): return (x >= a) and (x <= b)
>>> def pertPlano (x,y,x1,x2,y1,y2) : return pertLinear(x,x1,x2) and
pertLinear(y,y2,y1)
```

E o teste, para que valores interessam testar?

Etapa 4 [Analisando a Solução]: Existem outras maneiras de resolver o problema? Esta solução se aplica as outras dimensões?

EXERCÍCIOS

1. Avalie as expressões abaixo, explicando como foram reduzidas para o valor final, que é booleano. Se houver erro de avaliação, identifique a causa do erro.

   a) (3 < 4) or (5 == 7)

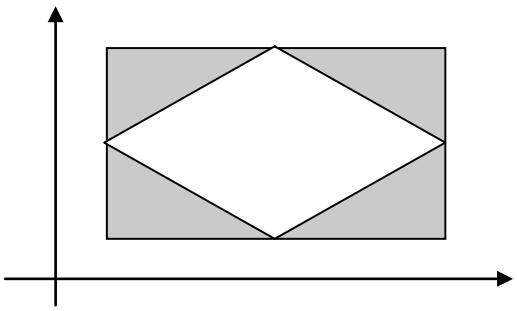
   b) not (3 < 7)

   c) (3+4 <= 8) and (10 != 34.7)

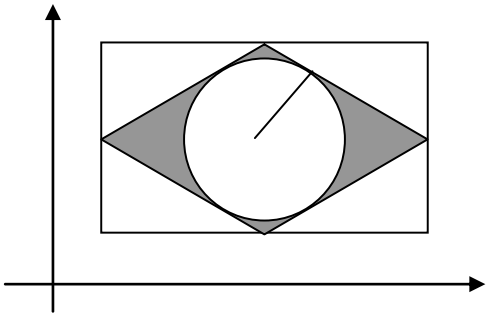
   d) (not (3 == 3)) or (not(4 != 5)) or (7 < 15) and (7 > 2)

2. Dado um ponto P(x,y) do plano cartesiano, defina funções que descrevam a sua pertinência nas regiões cinzas das figuras abaixo:

   i) A região R1 (região cinza), do retângulo dado pelas coordenadas do canto superior esquerdo e do canto inferior direito, como mostrado na figura abaixo:



   ii) A região R2 do losango (região cinza), sendo dados os pontos E e D do retângulo e sabendo-se que o círculo é tangente aos lados do losango.

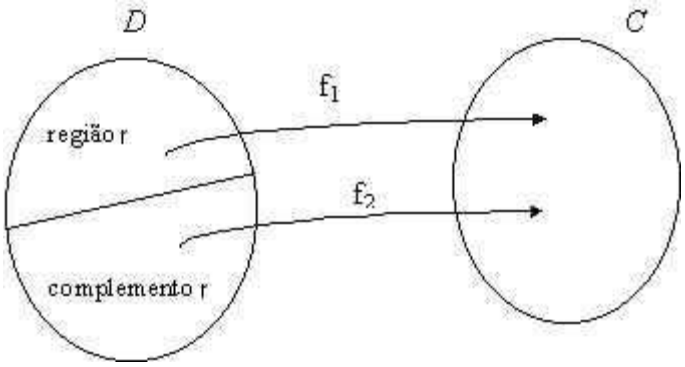


7. DEFINIÇÕES CONDICIONAIS

7.1. Introdução

Sabemos de nosso conhecimento matemático que algumas funções não são contínuas em um domínio e que, portanto, possuem várias definições.

Em muitos casos, o domínio D de uma função está dividido em regiões disjuntas que se complementam e, para cada uma dessas regiões, existe uma expressão que define o seu mapeamento no contra-domínio. Podemos representar esta situação pela figura abaixo:



Exemplo 1 - Considere a função que determina o valor da passagem aérea de um adulto, para um determinado trecho, por exemplo, Vitória-Manaus, considerando a sua idade. Pessoas com idade a partir de 60 anos possuem um desconto de 40% do valor. Considere ainda que a passagem para o trecho considerado custe R$ 600,00.

Temos aqui duas formas de calcular o valor da passagem de uma pessoa, dividindo o domínio em dois subconjuntos. O subconjunto dos adultos com menos de 60 anos e o subconjunto dos demais.

Podemos definir as duas funções a seguir:

```
>>> def valorPassagem1(): return 600
>>> def valorPassagem2(): return valorPassagem1() * 0.6
```

Para usar uma das definições, temos que explicitamente escolher a que se aplica ao nosso caso.

Exemplo 2 - Considere a função que associa com um determinado rendimento o Imposto de Renda a ser pago. Até um determinado valor, o contribuinte não paga imposto, e a partir de então o rendimento é dividido em faixas (intervalos), aos quais se aplicam diferentes taxas. Suponha a tabela hipotética abaixo.

Faixa	**alíquota**	**Desconto**
inferior ou igual a 10.800	0	0
entre 10.801 e 20.000	10	1000
entre 20.001 e 30.000	20	1500
acima de 30.000	25	1800

Para descrever as várias definições e os correspondentes subdomínios, poderíamos escrever separadamente cada definição, construindo, portanto várias funções, e deixar que o usuário escolha qual usar. Claro que isto traria muitos inconvenientes, pois estaríamos deixando uma escolha mecânica na mão do usuário, que além de sobrecarregá-lo com tarefas desnecessárias, ainda estaria expondo-o ao erro por desatenção. Mas vamos lá construir as funções independentes.

```
>>> def imposto1(s): return 0
>>> def imposto2(s): return s * 0.1 – 1000
>>> def imposto3(s): return s * 0.2 - 1500
>>> def imposto4(s): return s * 0.25 - 1800
```

Agora, para usá-las, o usuário pega o seu salário, olha a tabela e seleciona qual função aplicar.

A escolha de qual definição usar para uma dada situação é em si, um tipo de computação. Podemos descrever essa computação com expressões condicionais, deixando que o computador escolha. Descrevemos cada subdomínio com a respectiva função aplicável e deixemos que ele escolha a definição a aplicar, dependendo do valor fornecido. Vejamos então como isso pode ser feito nas seções subsequentes.

7.2. A estrutura IF-ELIF-ELSE

Vamos estudar inicialmente a construção mais simples de definição condicional, construída com a estrutua ***if-else*** segundo a seguinte sintaxe:

if <expressão lógica>**:** <expressão 1>
else: <expressão 2>

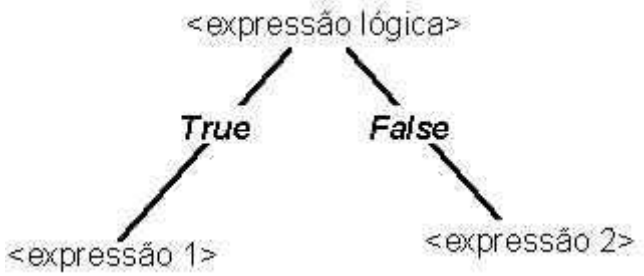
onde:

<expressão lógica>	Uma expressão descrevendo uma condição a ser satisfeita, envolvendo operadores relacionais e operadores lógicos.
<expressão1> e <expressão2>	1. Expressões descrevendo um valor a ser produzido como resposta à entrada fornecida e, como a expressão total deve ser de um único tipo, as duas expressões devem ser do mesmo tipo. 2. Cada uma destas expressões pode ser inclusive outra condicional, dentro da qual pode haver outras e assim sucessivamente. 3. Quando a <expressão lógica> é avaliada para ***True*** o valor resultante será o que for obtido pela avaliação da <expressão 1> caso contrário será o obtido pela avaliação da <expressão 2>

Para a função que calcula o valor da passagem aérea podemos então construir a seguinte definição:

```
>>> def valorPassagem(x):
        if x < 60: return 600
        else: return 360
```

Árvore de decisão: Podemos representar as expressões condicionais através de uma notação gráfica denominada de árvore de decisão. É importante considerar que este tipo de representação é uma ferramenta importantíssima para estruturarmos a solução de problemas que requerem expressões condicionais.



Exemplo 3 – Definir a função que determina o valor absoluto de um número. Sabemos que esta função se define em dois subdomínios:

subdomínio	**expressão**
x < 0	- x
x >= 0	x

Como só temos duas possibilidades, podemos codificar da seguinte maneira:

```
>>> def absoluto(x):
        if x < 0: return -x
        else: return x
```

Contudo, alguns problemas apresentam uma definição condicional mais complexa, de forma que o problema se divide em mais de dois subdomínios. Nestes casos, a estrutura ***if-elif-else***, complementar à recém-apresenta, deve ser usada segundo a seguinte sintaxe:

if <expressão lógica 1>**:** <expressão 1>
elif <expressão lógica 2>**:** <expressão 2>
...
elif <expressão lógica n-1>**:** <expressão n-1>
else: <expressão n>

Este é o caso do exemplo 2 que define a função para cálculo do Imposto de Renda. O domínio neste caso deve ser quebrado em quatro subdomínios e para cada um deles construiremos uma expressão.

Domínio	**Expressão**
s ≤ 10800	0
10800 < s ≤ 20000	s * 0.1 - 1000
20000 < s ≤ 30000	s * 0.2 - 1500
s > 30000	s * 0.25 - 1800

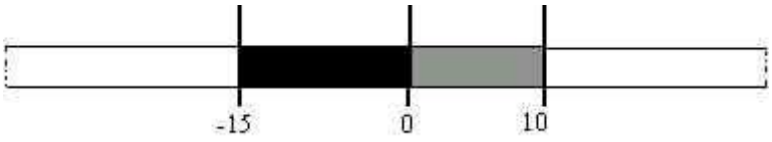
Para a dodificação, precisaremos quebrar sucessivamente o domínio da função em intervalos. A determinação dos intervalos é realizada da seguinte maneira: primeiro dividimos o domínio entre o primeiro intervalo e o restante, que por sua vez, é dividido entre o segundo intervalo e o seu restante e assim sucessivamente.

A codificação final pode ser:

```
>>> def imposto(s):
        if s <= 10800: return 0
        elif s <= 20000: return s * 0.1 - 1000
        elif s <= 30000: return s * 0.2 - 1500
        else: return s * 0.25 - 1800
```

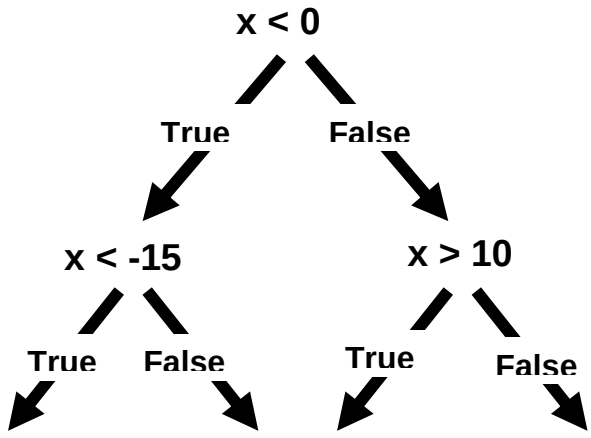
OUTROS EXEMPLOS

EXEMPLO 1: Considere um mapeamento de valores numéricos onde o domínio se divide em 4 regiões, cada uma das quais possui diferentes formas de mapeamento. As regiões são apresentadas na figura abaixo, numeradas da esquerda para direita. Observe ainda que as extremidades são abertas. Considere ainda a seguinte tabela de associações:



região	**mapeamento desejado**
região 1	o dobro de x
região 2	o sucessor de x
região 3	o quadrado de x
região 4	o simétrico do quadrado de x

Podemos analisar as regiões através do seguinte diagrama:

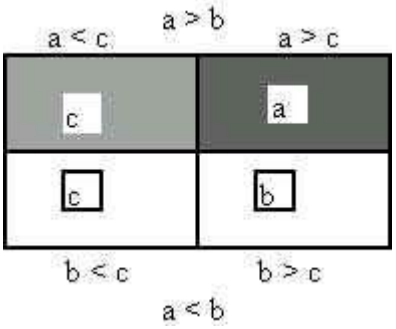


O que nos levará à seguinte definição:

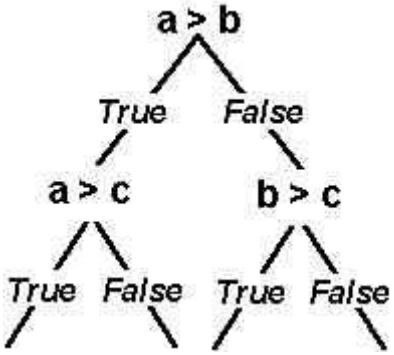
```
>>> def f(x):
        if x < 0:
             if x < -15: return 2*x
             else: return x + 1
        else:
             if x < 10: return x**2
             else: return - x**2
```

EXEMPLO 2: Dados três números inteiros distintos, determinar o maior deles.

Podemos explorar uma solução da seguinte maneira. Considere um retângulo e divida-o horizontalmente em 2 partes, a parte de cima representa as situações onde a > b e a de baixo aquelas onde b > a. Divida agora o retângulo verticalmente, em cada uma das regiões anteriores surgirão 2 metades. Na de cima, representamos agora a relação entre a e c. Na de baixo, a relação entre b e c.



Explorando as relações entre os números



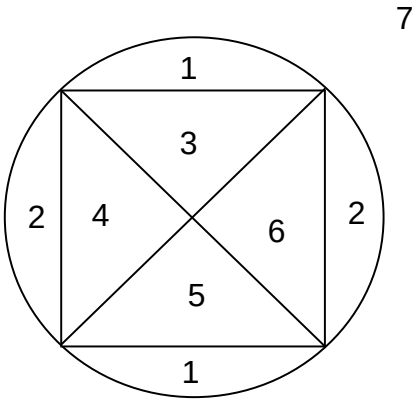
Representando as relações através de uma árvore de decisão

Traduzindo a árvore de decisão para Python, chegamos à seguinte definição:

```
>>> def maior(a, b, c):
        if a >= b:
             if a >= c: return a
             else: return c
        else:
             if b >= c: return b
             else: return c
```

EXERCÍCIOS

1. Sejam $C(x_1,y_1)$ o centro da circunferência de raio *r* e também do quadrado de lados paralelos aos eixos cartesianos e inscrito na circunferência. Escreva uma função que descreva a **região de pertinência do ponto P** na figura abaixo, dados $C(x_1,y_1)$, *r* e o ponto P (x,y),



8. O TESTE DE PROGRAMAS

8.1. Introdução

Não basta desenvolver um programa para resolver um dado problema. É preciso garantir que a solução esteja correta. Muitos erros podem ocorrer durante o desenvolvimento de um programa e, portanto temos que garantir que o programa que irá ser executado está livre de todos eles. Ao conceber a solução podemos nos equivocar e escolher caminhos errados. Precisamos eliminar esses equívocos. Ao codificarmos a nossa solução podemos cometer outros erros ao não traduzirmos corretamente nossa intenção. Esses erros podem ocorrer por um mau entendimento dos elementos da linguagem ou até mesmo por descuido, o certo é que eles ocorrem. Uma estratégia muito útil, mas não infalível, é o teste de programa.

Em sua essência, o teste de programa consiste em submeter um programa ao exercício de algumas instâncias do problema e comparar os resultados esperados com os resultados obtidos.

8.2. O Processo de Teste

Em primeiro lugar devemos escolher as instâncias apropriadas, não basta escolhê-las aleatoriamente. A seguir devemos determinar, sem o uso do programa, qual o valor que deveria resultar quando o programa for alimentado com essas instâncias. O passo seguinte consiste em submeter cada instância ao programa e anotar o resultado produzido por ele. Finalmente devemos comparar cada valor esperado com o valor produzido e descrever qual o tipo de ocorrência.

Um exemplo: Considere o problema de identificar se um dado ponto está ou não localizado no primeiro quadrante do espaço cartesiano. Considere ainda a seguinte definição:

```
>>> def primQuad(x, y): return (x >= 0) and (y >= 0)
```

Precisamos agora verificar se ela atende nossa intenção. Para tanto devemos escolher algumas instâncias, prever o resultado esperado e em seguida submeter ao intepretador, para ver o que acontece.

Que pares de valores deveremos escolher? Bom, vamos escolher uns pares usando a seguinte estratégia: um par onde x é maior que y, outro onde y seja maior que x e um terceiro em que os dois sejam iguais. Gerando uma planilha como apresentada a seguir.

x	y	resultado esperado	resultado obtido	diagnóstico
-5	-2	False		
-2	5	False		
5	5	True		

Podemos agora submeter as instâncias à avaliação do sistema, obtendo a seguinte interação:

```
>>> primQuad(-5, -2)
```

```
False
>>> primQuad(-2, 5)
False
>>> primQuad(5, 2)
True
```

Podemos agora completar o preenchimento de nossa planilha, obtendo a tabela a seguir:

x	y	resultado esperado	resultado obtido	diagnóstico
-5	-2	False	False	sucesso
-2	5	False	False	sucesso
5	2	True	True	sucesso

Verificando as diversas linhas da coluna “Diagnóstico”, parece que nosso programa está correto. Veja que ele passou com sucesso em todos os testes!

Que pena que não seja verdade. Apesar de passar em todos os testes a que foi submetido, ele não funciona corretamente. Tudo que podemos afirmar neste instante é que para os valores usados, o programa funciona corretamente. E para os outros valores, será que funciona corretamente? Outros valores? Quais?

8.3. Plano de Teste:

Para escolher os valores que usaremos, antes de mais nada, devemos identificar as classes de valores que serão relevantes para o teste, em um segundo instante podemos então escolher os representantes destas classes. Quando temos um parâmetro, os possíveis valores a serem usados são todas as constantes do domínio. Para o caso de um parâmetro do tipo ***int,*** existem 65536 valores diferentes. Testar nosso programa para todos esses valores implicaria em determinar a mesma quantidade de resultados esperados e em seguida digitar e submeter esta mesma quantidade de instâncias do problema para o sistema e ainda depois conferir um a um os resultado obtidos com os esperados. Enfim, um trabalho imenso. Imagine agora se fossem dois parâmetros? A quantidade de pares seria da ordem de 4.29497×10^9 (algo da ordem de quatro bilhões). E para 3 parâmetros? E para ***n*** parâmetros? Números cada vez mais enormes. Por este caminho, testar programas seria inviável.

Felizmente não precisamos de todos eles, basta identificar as classes distintas que importam para o problema, ou seja, as classes de equivalência relevantes. Isto pode ser obtido analisando o enunciado estendido do problema.

No caso de nosso exemplo anterior, analisando melhor a definição, podemos identificar que por definição, o espaço cartesiano se divide em quatro regiões. A primeira, onde ambos as coordenadas são positivas, denominamos de primeiro quadrante. A segunda, onde a coordenada x é negativa e a y positiva, denominamos de segundo quadrante. O terceiro quadrante corresponde ao espaço onde ambas as coordenadas são negativas. Ainda temos o quarto quadrante onde a coordenada x é positiva e a y é negativa. E os pontos que ficam em um dos eixos ou na interseção destes, qual a classificação que eles têm? Bom, podemos convencionar que não estão em nenhum dos 4 quadrantes descritos. Resumindo, nosso plano de teste deve contemplar estas situações, conforme é ilustrado na tabela a seguir.

x	y	quadrante
positivo	positivo	primeiro
negativo	positivo	segundo
negativo	negativo	terceiro
negativo	positivo	quarto
nulo	qualquer não nulo	eixo das ordenadas
qualquer não nulo	nulo	eixo das abscessos
nulo	nulo	origem

É bom observar ainda que este plano pode e deve ser preparado antes mesmo de elaborar o programa, para fazê-lo, precisamos apenas da especificação detalhada. Além disso, este plano não precisa ser feito pelo programador responsável pela elaboração da solução, qualquer outro programador, de posse do enunciado detalhado, pode se encarregar da tarefa. Este tipo de plano serve para alimentar o teste denominado de ***caixa preta***. Esta denominação se deve ao fato de não ser necessário conhecermos a estrutura da solução para elaborar o plano. Outro aspecto positivo da elaboração do plano o mais cedo possível é que contribuí para um melhor entendimento do problema.

Voltando ao nosso exemplo, podemos agora elaborar a nossa planilha de teste considerando as classes de equivalência a serem definidas. Uma questão que surge é como escolhemos o representante de uma classe? Existem melhores e piores? No nosso caso, como pode ser qualquer valor positivo ou negativo, podemos escolher valores de um dígito apenas, no mínimo escreveremos e digitaremos menos.

x	y	resultado esperado	resultado obtido	diagnóstico
2	3	True		
-2	3	False		
-2	-3	False		
2	-3	False		
0	3	False		
0	-3	False		
2	0	False		
-2	0	False		
0	0	False		

8.4. Realizando o Teste:

Vejamos agora o resultado de nossa interação com o interpretador.

```
>>> primQuad(2, 3)
True
>>> primQuad(-2, 3)
False
>>> primQuad(-2, -3)
False
>>> primQuad(2, -3)
False
>>> primQuad(0, 3)
True
```

```
>>> primQuad(0, -3)
False
>>> primQuad(2, 0)
True
>>> primQuad(-2, 0)
False
>>> primQuad(0, 0)
True
```

Voltemos agora para nossa planilha e vamos preenchê-la na coluna de resultado obtido e diagnóstico.

x	y	resultado esperado	resultado obtido	diagnóstico
2	3	True	True	sucesso
-2	3	False	False	sucesso
-2	-3	False	False	sucesso
2	-3	False	False	sucesso
0	3	False	True	falha
0	-3	False	False	sucesso
2	0	False	True	falha
-2	0	False	False	sucesso
0	0	False	True	falha

8.5. Depuração:

Uma vez testado o programa e identificado que ocorreram instâncias para as quais a nossa definição não está fazendo a associação correta, ou seja, o valor obtido é diferente do esperado, devemos passar a uma nova fase. Depurar um programa é um processo que consiste em buscar uma explicação para os motivos da falha e posteriormente corrigi-la. Obviamente isto também não é um processo determinante e nem possuímos uma fórmula mágica. Ao longo de nossa formação de programadores iremos aos poucos incorporando heurísticas que facilitem esta tarefa. Por enquanto é muito cedo para falarmos mais do assunto e vamos concluir com um fechamento do problema anterior. Após concluir as modificações devemos testar o programa novamente.

Depurando nossa solução - Podemos concluir por simples inspeção da nossa última planilha (aquela com todas as colunas preenchidas) que nossa solução está incorreta. Uma interpretação dos resultados nos leva à hipótese de que a nossa solução considera que quando o ponto se localiza na origem ou em um dos eixos positivos, a nossa definição está considerando que eles estão no primeiro quadrante.

Passo seguinte, verificar se de fato nossa definição incorre neste erro. Em caso afirmativo, corrigi-la e a seguir, resubmetê-la aos testes. Observando a nossa definição inicial, podemos concluir que de fato nossa hipótese se confirma.

```
>>> def primQuad(x, y): return (x >= 0) and (y >= 0)
```

Podemos então modificá-la para obter uma nova definição, que esperamos que esteja correta.

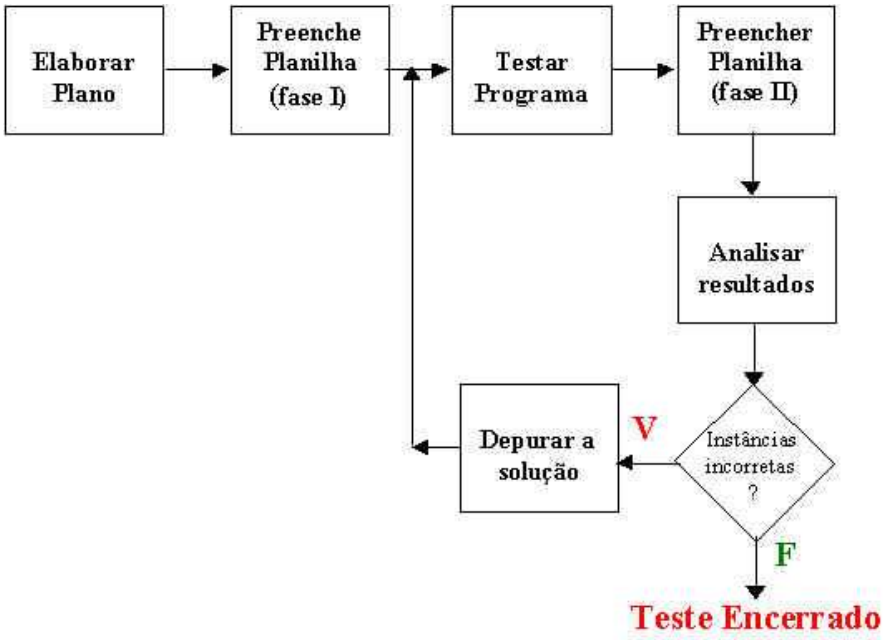
```
>>> def primQuad(x, y): return (x > 0) and (y > 0)
```

Agora é submetê-la novamente ao teste e ver o que acontece!

8.6. Uma Síntese do Processo:

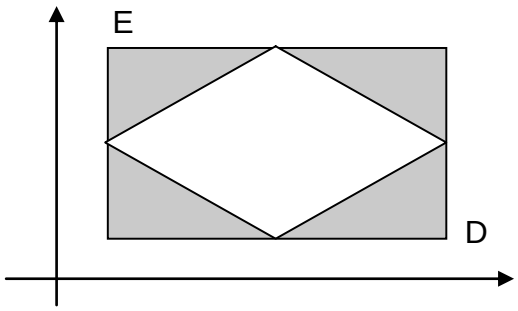
O processo é, como vimos, repetitivo e só se encerra quando não identificarmos mais erros. O diagrama ilustra o processo como um todo.

Mas lembre-se, isto ainda não garante que seu programa esteja 100% correto! Quando não identificamos erro, apenas podemos concluir que para as instâncias que usamos o nosso programa apresenta os resultados esperados.



EXERCÍCIO:

1. Dado um ponto P(x,y) do plano cartesiano, defina funções que descrevam a sua pertinência na região cinza da figura abaixo (a região R1 - região cinza - do retângulo dado pelas coordenadas do canto superior esquerdo e do canto inferior direito. Faça o plano de teste para o seu programa:



9. RESOLVENDO PROBLEMAS - OS MOVIMENTOS DO CAVALO

9.1. Introdução

Considere o jogo de xadrez, onde peças são movimentadas em um tabuleiro dividido em 8 linhas e oito colunas. Considere ainda os movimentos do cavalo, a partir de uma dada posição, conforme diagrama a seguir, onde cada possível movimento é designado por ***mi*.** No esquema, o cavalo localizado na posição (5, 4) pode fazer oito movimentos, onde o primeiro deles, ***m1***, levaria o cavalo para a posição (7,5). Por sua vez, o cavalo localizado na posição (1,8) pode realizar apenas os movimentos m5 e m6, sendo que o movimento m5 leva-o para a posição (2,6).

	1	2	3	4	5	6	7	8
1								C
2						m5		
3			m3		m2		m6	
4		m4				m1		
5				C				
6		m5				m8		
7			m6		m7			
8								

9.2. Problema 1:

Escreva uma função que determina se, a partir de uma dada posição, o cavalo pode ou não realizar o primeiro movimento. Vamos chamá-la de `m1EhPossivel`, e denominar seus parâmetros (a posição corrente), de **`linha`** e **`coluna`**. Eis alguns exemplos de uso de nossa função e os valores esperados:

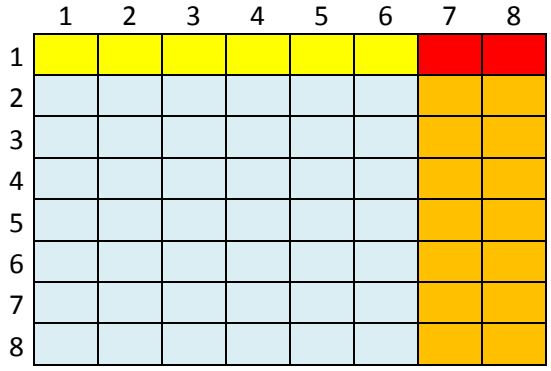
Instância	**resultado**
m1EhPossivel(5, 4)	True
m1EhPossivel(8, 1)	False
m1EhPossivel(1, 1)	True
m1EhPossivel(1, 8)	False

9.2.1. Solução - A solução pode ser encontrada através da construção de uma expressão booleana que avalie se a nova posição, ou seja, aquela em que o cavalo seria posicionado pelo primeiro movimento, está dentro do tabuleiro. Como o cavalo, no primeiro movimento, anda duas casas para direita e uma para cima, basta verificar se as coordenadas da nova posição não ultrapassam a oitava fileira (linha ou coluna).

Codificando, temos então:

```
>>> def m1EhPossivel(linha, coluna):
        return (linha + 2 <= 8) and (coluna + 1 <= 8)
```

9.2.2. Testando a Solução - como já discutimos anteriormente, para verificar a correção de nossa solução, devemos submetê-la a um teste. Para tanto devemos escolher as classes relevantes e, a partir delas, produzir a nossa planilha de teste. Olhando para a especificação do problema, podemos identificar 4 classes que se equivalem para os fins do nosso programa, como podemos observar na figura abaixo:



Assim, a seguinte tabela de classes pode ser estabelecida para o problema em questão:

região	**linha**	**coluna**	**resultado**
amarela	1	qualquer	m1 não é possível
laranja	qualquer	7 ou 8	m1 não é possível
azul	2 a 8	1 a 6	m1 é possível

Observe que a região em vermelho não tem uma classe específica, porém é contemplada tanto pela amarela quanto pela laranja.

Fica como sugestão de exercício fazer a planilha de testes e aplicar os testes.

9.2.3. Estendendo o Problema - Podemos fazer o mesmo para todos os demais movimentos, obtendo com isso as seguintes definições:

```
>>> def m1EhPossivel(linha, coluna):
        return (linha-1>=1) and (coluna+2<=8)
>>> def m2EhPossivel(linha, coluna):
        return (linha-2>=1) and (coluna+1<=8)
>>> def m3EhPossivel(linha, coluna):
        return (linha-2>=1) and (coluna-1>=1)
>>> def m4EhPossivel(linha, coluna):
        return (linha-1>=1) and (coluna-2>=1)
>>> def m5EhPossivel(linha, coluna):
        return (linha+1<=8) and (coluna-2>=1)
>>> def m6EhPossivel(linha, coluna):
        return (linha+2<=8) and (coluna-1>=1)
>>> def m7EhPossivel(linha, coluna):
        return (linha+2<=8) and (coluna+1<=8)
>>> def m8EhPossivel(linha, coluna):
        return (linha+1<=8) and (coluna+2<=8)
```

9.2.4. Identificando Abstrações - Podemos agora indagar, olhando para as oito definições, sobre a ocorrência de algum conceito geral que permeie todas elas. Poderíamos também ter feito isso antes. Como não o fizemos, façamo-lo agora. Podemos observar que todas elas avaliam duas expressões e que ambas testam fronteiras que podem ser margem direita, margem esquerda, margem superior ou margem inferior. Podemos observar ainda, que o par margem direita e margem superior testam o mesmo valor 8, assim como ocorre com as duas outras, que testam o valor 1. Com isso podemos definir duas novas funções, **`testaLimiteMax(x)`** e **`testaLimiteMin(x)`**, para testar estes limites. Agora, as nossas definições anteriores podem ser reescritas, usando as duas abstrações identificadas.

```
>>> def testaLimiteMax(x): return (x <= 8)
>>> def testaLimiteMin(x): return (x >= 1)
>>> def m1EhPossivel(linha, coluna):
     return testaLimiteMin(linha-1) and testaLimiteMax(coluna+2)
>>> def m2EhPossivel(linha, coluna):
     return testaLimiteMin(linha-2) and testaLimiteMax(coluna+1)
>>> def m3EhPossivel(linha, coluna):
     return testaLimiteMin(linha-2) and testaLimiteMin(coluna-1)
>>> def m4EhPossivel(linha, coluna):
     return testaLimiteMin(linha-1) and testaLimiteMin(coluna-2)
>>> def m5EhPossivel(linha, coluna):
     return testaLimiteMax(linha+1) and testaLimiteMin(coluna-2)
>>> def m6EhPossivel(linha, coluna):
     return testaLimiteMax(linha+2) and testaLimiteMin(coluna-1)
>>> def m7EhPossivel(linha, coluna):
     return testaLimiteMax(linha+2) and testaLimiteMax(coluna+1)
>>> def m8EhPossivel(linha, coluna):
     return testaLimiteMax(linha+1) and testaLimiteMax(coluna+2)
```

9.2.5. Análise da Solução - O que será que ganhamos com esta nova forma de descrever a nossa solução? Podemos indicar pelo menos três indícios de vantagem na nova solução:

1. Clareza - Na medida em que agora está explicitado, que todas as oito funções para verificar os movimentos possuem estrutura semelhante e que todas estão usando funções para verificar a ultrapassagem das bordas;
2. Manutenção - Se nosso tabuleiro mudasse, ou seja, passasse a ter 9 linhas por nove colunas, bastaria alterar a função **`testaLimiteMax(x)`** e tudo estaria modificado, ao invés de termos que alterar as oito definições.
3. Reuso - As duas funções que testam as bordas poderiam ser usadas para construir funções para avaliar o movimento de outras peças do jogo de xadrez.

9.3. Problema 2:

Sabemos que para cada posição alguns movimentos podem ser realizados e outros não. Como ordenamos os movimentos no sentido anti-horário, gostaríamos de obter, para uma dada posição, dos movimentos que podem ser realizados, aquele que possui o menor índice. Vejamos o esquema a seguir.

	1	2	3	4	5	6	7	8
1	m4				m1			c1
2			c3			m5		
3	m5				m8		m6	
4		m6		m7				
5								
6		m2					m3	
7			m1			m4		
8	c4							c2

Podemos observar que o cavalo C1 só pode fazer os movimentos m5 e m6 e que o de menor índice é m5. Já o cavalo C2 só pode fazer os movimentos m3 e m4 e que o de menor índice é o m3. Enquanto isso o cavalo C3 pode fazer os movimentos m1, m4, m5, m6, m7 e m8. Para este caso o movimento de menor índice é o m1.

Vamos chamar esta função de `movimentaCavalo` e, como no problema anterior, os parâmetros serão as coordenadas da posição atual do cavalo. Eis alguns exemplos de uso de nossa função:

Instância	resultado
movimentaCavalo(1, 8)	5
movimentaCavalo(8, 8)	3
movimentaCavalo(2, 3)	1
movimentaCavalo(8, 1)	1

9.3.1. Solução - Como já sabemos, para verificar se um dado movimento ***mi*** é possível, basta ter um meio de verificar um-a-um os movimentos, a partir do primeiro (m1) e encontrar o primeiro que pode ser realizado. Quando isso ocorrer podemos fornecer como resposta o seu índice. Podemos construir para isso uma árvore de decisão. Na raiz da árvore estará a pergunta

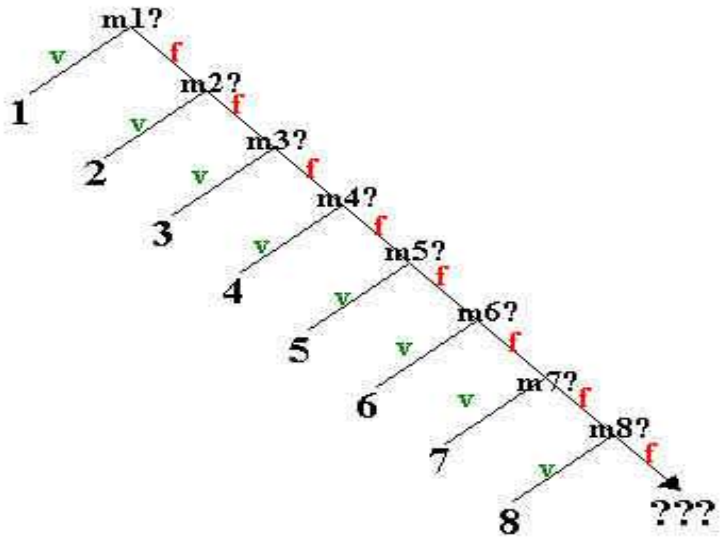
"é possível realizar o movimento m1?"

Em caso afirmativo (braço esquerdo da árvore), mapeamos no valor 1 e em caso negativo (braço direito), o que devemos fazer? Bom, aí podemos começar uma nova árvore (na verdade uma sub-árvore), cuja raiz será:

"é possível realizar o movimento m2?"

E daí, prosseguimos até que todos os movimentos tenham sido considerados.

A árvore resultante será:



9.3.2. Codificando a Solução - Vamos então explorar os recursos da linguagem para transformar nosso plano em um programa que de fato possa ser "entendido" pelo nosso sistema de programação. Como podemos observar temos aqui o caso de uma função que não é contínua para o domínio do problema. Pelo que sabemos até então, não dá para expressar a solução como uma única expressão simples. Resta-nos o recurso das expressões condicionais. Para verificar se um dado movimento é satisfeito podemos usar as funções que construímos anteriormente e com isso obtemos a seguinte definição:

```
>>> def movimentaCavalo(linha, coluna):
        if m1EhPossivel(linha, coluna): return 1
        elif m2EhPossivel(linha, coluna): return 2
        elif m3EhPossivel(linha, coluna): return 3
        elif m4EhPossivel(linha, coluna): return 4
        elif m5EhPossivel(linha, coluna): return 5
        elif m6EhPossivel(linha, coluna): return 6
        elif m7EhPossivel(linha, coluna): return 7
        elif m8EhPossivel(linha, coluna): return 8
        else: return 0
```

9.3.3. Análise da Solução - Em primeiro lugar, incluímos a resposta igual a zero (0) quando o movimento m8, o último a ser avaliado, resulta em fracasso. Para que serve isso? Acontece que se a posição de entrada não for válida, ou seja, alguma das coordenadas não pertencer ao intervalo [1, 8], nenhum movimento seria válido e se não providenciarmos uma resposta alternativa, nossa função seria parcial. Mas isto resolve de fato nosso problema? O que ocorreria se a posição de entrada fosse (0, 0)? Bom, nossa função determinaria que o sétimo movimento poderia ser realizado e isto não é verdade. A “invenção” de um resultado extra para indicar que não há solução possível, transformando uma função parcial em uma função total, é razoável, mas como foi feita não resolveu.

Uma solução é preceder a tentativa de determinar a solução adequada por uma avaliação da validade dos dados de entrada. Neste caso, bastaria verificar se os dois estão no intervalo [1, 8]. Podemos construir aqui uma função que avalia a pertinência de um valor a um intervalo numérico, ou, mais especificamente, uma função que verifica a pertinência no tabuleiro, conforme definições a seguir:

```
>>> def pert(x, a, b): return (x >= a) and (x <= b)

>>> def pertTabuleiro(linha, coluna):
        return pert(linha, 1, 8) and pert(coluna, 1, 8)
```

Especulando um pouco mais sobre a nossa solução, podemos observar que o movimento ***m8***, jamais ocorrerá! Analisando os possíveis movimentos chegaremos à conclusão de que para cada posição há pelo menos 2 movimentos possíveis, assim, o oitavo movimento nunca é o único movimento possível. Sugerimos fortemente que o leitor prove este teorema. Portanto a solução final pode ser:

```
>>> def movimentaCavalo(linha, coluna):
        if not pertTabuleiro(linha,coluna): return 0
        elif m1EhPossivel(linha, coluna): return 1
        elif m2EhPossivel(linha, coluna): return 2
        elif m3EhPossivel(linha, coluna): return 3
        elif m4EhPossivel(linha, coluna): return 4
        elif m5EhPossivel(linha, coluna): return 5
        elif m6EhPossivel(linha, coluna): return 6
        else: return 7
```

9.4. Revisitando o Problema 1:

Observando a solução encontrada para o problema 1, constatamos que embora a noção de movimento do cavalo seja única, quem precisar saber se um dado movimento é válido, precisará conhecer o nome das oito funções. Embora seja cedo para falarmos de interface homem-máquina, já dá para dizer que estamos sobrecarregando nosso usuário ao darmos oito nomes para coisas tão parecidas. Será que temos como construir uma só função para tratar o problema? Vamos reproduzir aqui a interface das oito:

```
m1EhPossivel(linha, coluna)
m2EhPossivel(linha, coluna)
m3EhPossivel(linha, coluna)
m4EhPossivel(linha, coluna)
m5EhPossivel(linha, coluna)
m6EhPossivel(linha, coluna)
m7EhPossivel(linha, coluna)
m8EhPossivel(linha, coluna)
```

Propositadamente escrevemos o nome delas com um pedaço em vermelho e outro em preto. Seria alguma homenagem à algum time que tem essas cores? Na verdade estamos interessados em destacar que a pequena diferença nos nomes sugere que temos uma mesma função e que existe um parâmetro oculto. Que tal explicitá-lo? Podemos agora ter uma função com 3 parâmetros, sendo o primeiro deles para indicar o número do movimento que nos interessa. A interface agora seria:

```
movEhPossivel(mov, linha, coluna)
```

Agora, por exemplo, para solicitar a avaliação do sétimo movimento para um cavalo em (3, 4), escrevemos:

```
>>> movEhPossivel(7, 3, 4)
True
```

Muito bem, e como codificaremos isso?

9.4.1. Solução - Precisamos encampar em nossa solução o fato de que a nossa função possui diferentes formas de avaliação, para diferentes valores do domínio, algo parecido com a solução do problema 2, ou seja, a nossa função não é continua e portanto temos que selecionar qual a definição apropriada para um determinado valor de m. Devemos construir uma árvore de decisão. Aqui deixamos esta tarefa a cargo do leitor e passamos direto à codificação conforme apresentamos a seguir:

```
>>> def testaLimiteMax(x): return (x <= 8)
>>> def testaLimiteMin(x): return (x >= 1)
>>> def movEhPossivel(numMov, linha, coluna):
     if not pertTabuleiro(linha,coluna): return False
     if numMov == 1: return testaLimiteMin(linha-1) and testaLimiteMax(coluna+2)
     if numMov == 2: return testaLimiteMin(linha-2) and testaLimiteMax(coluna+1)
     if numMov == 3: return testaLimiteMin(linha-2) and testaLimiteMin(coluna-1)
     if numMov == 4: return testaLimiteMin(linha-1) and testaLimiteMin(coluna-2)
     if numMov == 5: return testaLimiteMax(linha+1) and testaLimiteMin(coluna-2)
     if numMov == 6: return testaLimiteMax(linha+2) and testaLimiteMin(coluna-1)
     if numMov == 7: return testaLimiteMax(linha+2) and testaLimiteMax(coluna+1)
     if numMov == 8: return testaLimiteMax(linha+1) and testaLimiteMax(coluna+2)
     return False
```

Nesta solução, utilizamos uma característica da linguagem que interfere no uso da estrutura if-elif-else. Observe que utilizamos apenas **`if's`** para codificar a solução acima. A estrutura supracitada torna-se desnecessária quando as condições precedem definições de funções pois, uma vez que a condição é verdadeira, a definição correspondente será aplicada e o restante das condições não será verificado. Por exemplo, se a condição **`numMov == 1`** é verdadeira, as condições a seguir sequer serão realizadas. Por isso também, a definição return False ao final só será realizada se nenhuma das anteriores for verdade, ou seja, se o movimento informado não estiver no intervalo de 1 a 8.

9.4.2. Análise da Solução - Ao contrário da solução do problema 2, onde necessariamente a validade dos movimentos tinha que ser verificada do primeiro para o último, pois nos interessava saber qual o primeiro possível de ser realizado, o leitor pode observar que esta ordem aqui não é necessária. Tanto faz se perguntamos primeiro se m=7 ou se m=3. Será que podemos tirar algum proveito disso? Alertamos o iniciante, que devemos sempre identificar propriedades internas do problema e explorá-las adequadamente. Qual a influência desta ordem na eficiência da avaliação de uma expressão submetida ao interpretador? Para responder, basta lembrar que as condições são avaliadas sequencialmente. Por exemplo, se a posição está fora do tabuleiro faremos uma verificação, caso contrário, se m=8, teremos que avaliar 9 condições, se m=1 faremos duas avaliações e se m está fora do domínio teremos feito também nove avaliações. Ou seja, no pior caso faremos 9 avaliações e no melhor caso uma (1). Em média, ao longo do uso da função, assumindo uma distribuição uniforme dos valores de m, faremos 4 avaliações. E se o intervalo fosse de 100 elementos distintos, quantas avaliações de condições faríamos em média? Será possível elaborar soluções onde este número de avaliações seja menor?

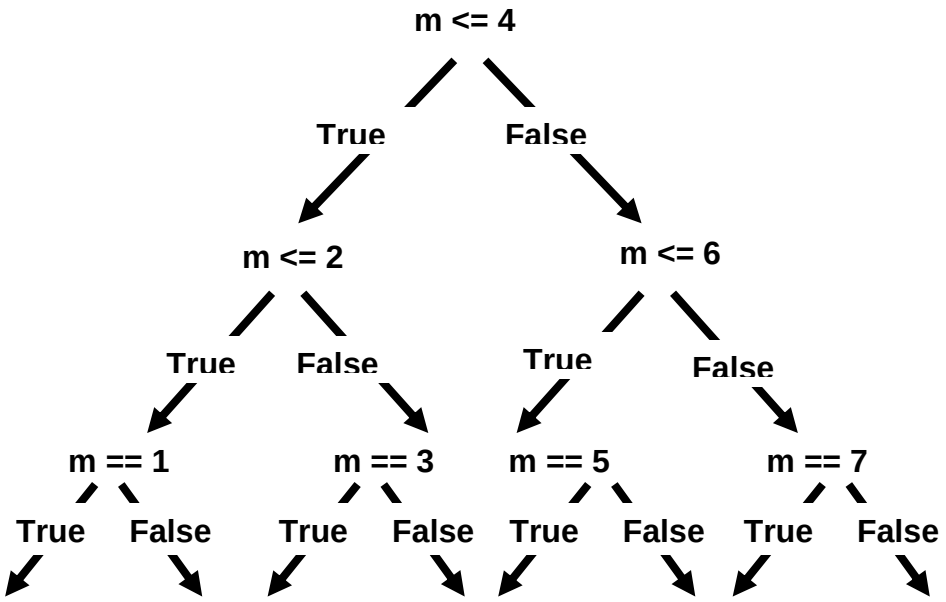
A resposta é sim! Já que a ordem das avaliações não importa, podemos buscar uma ordem mais conveniente para reduzir o número de avaliações por cada instância avaliada.

A idéia geral para estes casos é obtida a partir de um conceito de vasta utilidade na Computação. Falamos de ***árvore binária*** e o leitor por certo ouvirá falar muito dela ao longo da vida enquanto profissional da área.

O caminho consiste em dividir o intervalo considerado ao meio e fazer a primeira pergunta, por exemplo, m <= 4? Dividindo em 2 o intervalo de comparações, para cada um destes podemos novamente dividir ao meio, até que não mais tenhamos o que dividir, como ilustramos no diagrama a seguir, onde cada linha representa um passo da divisão dos intervalos.

<1	1	2	3	4	5	6	7	8	>8
<1	1	2	3	4	5	6	7	8	>8
<1	1	2	3	4	5	6	7	8	>8

Que também podemos representar por:



A codificação ficará então:

```
def movEhPossivel(numMov, linha, coluna):
    if not (1 <= numMov <= 8): return False
    if not pertTabuleiro(linha, coluna): return False
    if numMov <= 4:              # numMov <=4
        if numMov <= 2:          # 1 <= numMov <=2
            if numMov == 1:      # numMov == 1
              return pertTabuleiro(linha-1,coluna+2)
            else:                # numMov == 2
              return pertTabuleiro(linha-2,coluna+1)
        else:                    # 2 < numMov <=4
            if numMov == 3:      # numMov == 3
              return pertTabuleiro(linha-2,coluna-1)
            else:                # numMov == 4
              return pertTabuleiro(linha-1,coluna-2)
    else:                        # 4 < numMov <= 8
```

```
        if numMov <= 6:          # 4 < numMov <= 6
            if numMov == 5:      # numMov == 5
              return pertTabuleiro(linha+1,coluna-2)
            else:                # numMov == 6
              return pertTabuleiro(linha+2,coluna-1)
        else:                    # 6 < numMov <= 8
            if numMov == 7:      # numMov == 7
              return pertTabuleiro(linha+2,coluna+1)
            else:                # numMov == 8
              return pertTabuleiro(linha+1,coluna+2)
```

Se fizermos uma análise das possibilidades veremos que para qualquer valor de m o número máximo de avaliações que faremos será sempre igual a quatro (ou seis, contanto as duas verificações iniciais de validação dos dados)! E se fosse 100 valores para m, qual seria o número máximo de comparações? E para 1000? E 1000000?

O número de comparações, seguindo este esquema, é aproximadamente igual ao logaritmo do número de valores na base 2. Portanto para 100 teríamos 7, para 1000 teríamos 10 e para um milhão teríamos 20. Veja que é bem inferior ao que obtemos com o esquema anterior, também denominado de linear. Confira a comparação na tabela abaixo.

número de valores	**esquema linear** (número médio)	**esquema binário** (número máximo)
8	4	4
100	50	7
1000	500	10
1000000	500000	20

EXERCÍCIOS:

1. Escreva uma função que verifica se, dados um movimento válido (1 a 8) e a posição do cavalo, pode-se ou não realizar o movimento desejado.
2. Escreva uma função que determina, dados um movimento válido (1 a 8) e a posição do cavalo, em qual linha do tabuleiro estará situado o cavalo após a realização do movimento. Sua função deve se chamar **`movimentaCavaloL`** e uma instância dela seria **`movimentaCavaloL(1,4,5)`** $\Rightarrow$ **`5`**.
3. Faça uma função similar à anterior, mas agora sua função deve se chamar **`movimentaCavaloC`** e deve responder a coluna correspondente da nova posição do cavalo no tabuleiro, após a realização do movimento desejado. Uma instância dela seria **`movimentaCavaloC(1,4,5)`** $\Rightarrow$ **`7`**.

10. VALIDAÇÃO DE DADOS

Como sabemos, toda função tem um domínio e um contradomínio. Quando tentamos avaliar uma instância de uma definição, usando valores fora desse domínio, podemos receber como resposta mensagens nem sempre esclarecedoras. Quando se trata do usuário de nosso programa, esta situação se mostra mais indesejável ainda. Aqueles que constroem a definição podem discernir com mais facilidade a natureza dos problemas que ocorrem durante o seu desenvolvimento. O mesmo não se pode esperar de alguém que não tem conhecimento dos elementos internos de nossas definições. Neste caso, é interessante antever a exceção dar um tratamento adequado, como exibir uma mensagem discernível.

Tipicamente os erros serão de duas naturezas. Pode ser que o tipo da instância esteja correto, mas nossa definição use alguma função primitiva que não se aplica ao valor da instância. Por exemplo, se temos a seguinte definição de **`f`**, e a submetemos a uma instância onde ***y = 0***, teremos como resultado algo da seguinte natureza:

```
>>> def f(x, y, z): return x/y + z
>>> f(2,0,4)
Traceback (most recent call last):
  File "<pyshell#350>", line 1, in <module>
    f(2,0,4)
  File "<pyshell#349>", line 1, in f
    def f(x, y, z): return x/y + z
ZeroDivisionError: integer division or modulo by zero
```

Um outro tipo de problema ocorre quando o tipo de algum parâmetro da nossa instância não casa com o tipo inferido pelo interpretador. Por exemplo, se usamos um valor do tipo ***Char*** para ***x***, obtemos:

```
>>> f('a',2,3)
Traceback (most recent call last):
  File "<pyshell#351>", line 1, in <module>
    f('a',2,3)
  File "<pyshell#349>", line 1, in f
    def f(x, y, z): return x/y + z
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

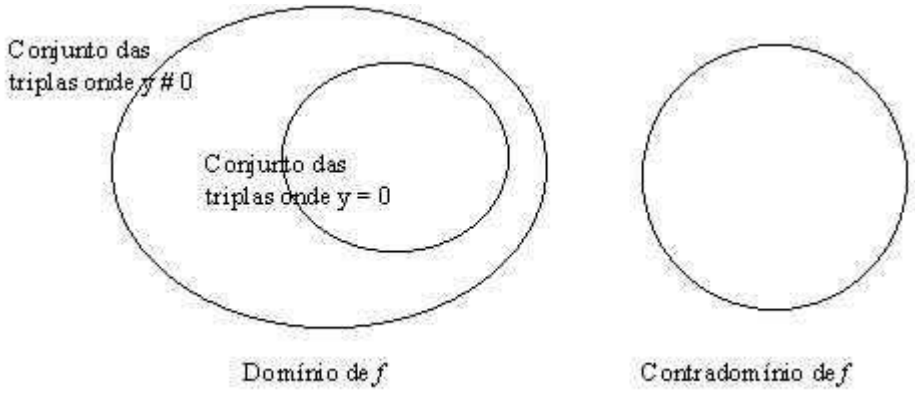
Nesta seção trataremos do primeiro caso, deixando o segundo para outra oportunidade.

10.1. Caracterizando a situação

Voltemos à definição anteriormente apresentada:

```
>>> def f(x, y, z): return x/y + z
```

Neste caso, os dois primeiros parâmetros são inferidos como sendo do tipo numérico**,** visto que os operadores / e + só se aplicam a valores deste tipo. Portanto o universo será formado por todos os possíveis ternos de valores do tipo numérico*.* Chamemo-lo de ***T***. Entretanto, o domínio de nossa função não é exatamente o conjunto de constantes de ***T***, posto que a função não está definida para as constantes de ***T*** cujo segundo elemento é zero.



Desejamos reconstruir a função original incluindo uma **condição de proteção** contra as violações de domínio. Para isso, precisamos restringir o domínio de **`f`** apenas para os casos em que o **`y`** é diferente de zero. A função é definida de forma condicional, como vimos anteriormente. Entretanto, ao invés de ter diferentes definições da função cada condição, a **condição de proteção** trata o(s) caso(s) para o(s) qual(is) a função não está definida.

Podemos simplesmente não calcular a expressão nesses casos excepcionais:

```
>>> def f(x, y, z):
        if y != 0: return x/y + z
>>> f(2,0,4)
```

ou ainda informar uma mensagem:

```
>>> def f(x, y, z):
        if y == 0: return "Divisao por zero nao permitida"
        else: return x/y + z
>>> f(2,0,4)
'Divisao por zero nao permitida'
```

Retornar uma mensagem parece ser a opção adequada para alertar o usuário quanto ao tipo de erro que está acontecendo. Contudo, se o objetivo da sua função é ser reutilizada em outra função, uma mensagem do tipo *string* não será devidamente entendida. Pelo contrário, se a função espera um resultado numérico com o qual realizará uma outra operação, a mensagem enviada para evitar um erro provocará outro. Veja o exemplo:

```
>>> from math import sqrt
>>> def g(x, y, z): return sqrt(f(x,y,z))

>>> g(2, 0, 4)
Traceback (most recent call last):
  File "<pyshell#85>", line 1, in <module>
    g(2,0,4)
  File "<pyshell#84>", line 1, in g
    def g(x,y,z): return sqrt(f(x,y,z))
TypeError: a float is required
```

Em outras palavras, o valor retornado para pela função **`f`** não foi um valor numérico, como a função **`g`** esperava para realizar a operação de raiz quadrada, mas uma *string*. Então, se

imprimíssemos a mensagem, usando a função **print**, ao invés de retorná-la como resultado da função, estaria resolvido o problema? Também não, vejamos o exemplo modificando-se a função **f**:

```
>>> from math import sqrt
>>> def f(x, y, z):
        if y == 0: print("Divisao por zero nao permitida")
        else: return x/y + z

>>> def g(x, y, z):
        return sqrt(f(x,y,z))

>>> g(2,0,4)
Divisao por zero nao permitida
Traceback (most recent call last):
  File "<pyshell#85>", line 1, in <module>
    g(2,0,4)
  File "<pyshell#84>", line 1, in g
    def g(x,y,z): return sqrt(f(x,y,z))
TypeError: a float is required
```

Embora agora a mensagem apareça para o usuário, o erro não está sendo de fato tratado, pois a função **f** ainda não retorna o tipo esperado pela função **g**. Na realidade, não há retorno algum, e, portanto, o erro permanece.

A maneira de se tratar esse problema em Python chama-se tratamento de exceção. A idéia é de que a execução da função tenha um curso normal e outro curso para casos excepcionais. Veremos aqui uma forma simplificada de usar este recurso, conforme mostra o exemplo:

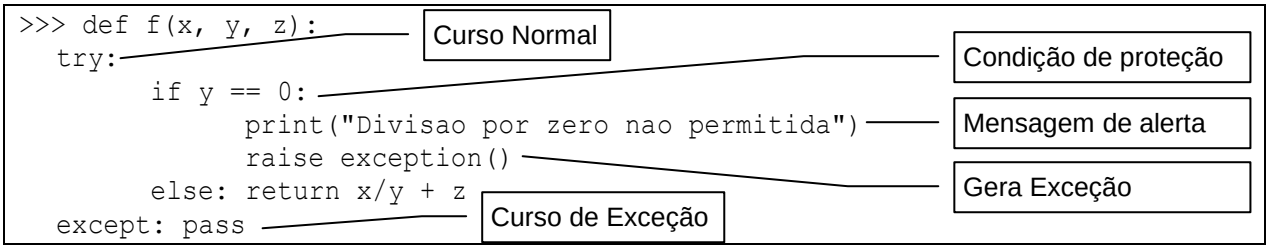
```
>>> from math import sqrt
>>> def f(x, y, z):
  try:
        if y == 0:
              print("Divisao por zero nao permitida")
              raise exception()
        else: return x/y + z
  except: pass

>>> def g(x, y, z):
  try:
        if f(x,y,z) < 0:
              print("Operacao de raiz quadrada não permitida")
              raise exception()
        else: return sqrt(f(x,y,z))
  except: pass

>>> g(2,0,4)
Divisao por zero nao permitida
>>> g(2,1,-4)
Operacao de raiz quadrada não permitida
```

Tanto a função **f** quanto a **g** tem um “curso normal” definido dentro de um bloco inciado pela palavra **try** e um “curso de exceção” definido dentro de um bloco inciado pela palavra **except**. Uma vez que a condição de proteção indica os casos para os quais a função não

tem definição, desejamos então interromper o processamento da função, idealmente avisando ao usuário o que aconteceu. Sendo assim, adotaremos o seguinte procedimento: imprime uma mensagem e em seguida gera uma exceção, que interrompe a execução do "curso normal" e segue o curso de Exceção. Neste curso, a palavra reservada **`pass`** indica simplesmente que nada há para ser feito.



Observe que, para este tratamento funcionar corretamente, tanto a função **`f`** quanto a **`g`** devem contemplar os cursos "normal" e "de exceção". Caso contrário, se, por exemplo, a função **`g`** não contempla tais cursos, a instanciação **`g(2,0,4)`** provocará um erro idêntico ao caso em que **`f`** apenas imprime a mensagem de alerta, sem retornar valor algum.

10.2. Um exemplo - raízes de uma equação do 2o. Grau

Voltemos ao problema de determinar as raízes de uma equação do segundo grau. Sabemos que o universo definido pelos tipos dos três parâmetros (a, b, c), é maior que o domínio da função. Ou seja, a função não está definida para instâncias de a, b e c, onde se verificam as seguintes condições:

```
a = 0 ou (b^2) + (4 * a * c) < 0
```

Podemos, então, redefinir a função que calcula uma das raízes da seguinte forma:

```
>>> from math import sqrt
>>> def raiz1eq2gr(a, b, c):
       def delta(): return b*b - 4*a*c
       if a == 0: return "Esta equacao nao eh de 2o grau"
       elif delta() < 0: return "Delta negativo!"
       else: return (-b + sqrt(delta()))/2*a
>>> raiz1eq2gr(3, 2, 1)
'Delta negativo!'
```

ou ainda:

```
>>> from math import sqrt
>>> def raiz1eq2gr(a, b, c):
        try:
              def delta(): return b*b - 4*a*c
              if a == 0:
                    print("Esta equacao nao eh de 2o grau")
                    raise exception()
              elif delta() < 0:
                    print("Delta negativo!")
                    raise exception()
              else: return (-b + sqrt(delta()))/2*a
        except: pass
>>> raiz1eq2gr(3,2,1)
Delta negativo!
```

EXERCÍCIOS

1. Refaça os exercícios do capítulo 9, verificando se as posições ou movimento informados são válidos.
2. Faça uma função calcular 1/(x+ y) tratando possíveis casos de erro.
3. Forneça dois exemplos de funções que não tem definição para algum valor do domínio.

11. TUPLAS

11.1. Introdução

Até então trabalhamos com valores elementares. Ou seja, valores atômicos, indivisíveis no que diz respeito às operações de seu tipo de dados. Por exemplo, ***True***, ***523***, ***8.234***, são valores elementares dos tipos Boolean, Integer e Float, respectivamente. Por exemplo, dentro da linguagem não podemos nos referir ao primeiro algarismo do valor ***523***. Dizemos que esses são tipos primitivos da linguagem. O elenco de tipos primitivos de uma linguagem pode ser entendido como o alicerce da linguagem para a descrição de valores mais complexos, que são denominados genericamente de ***valores estruturados***.

Em geral, os problemas que pretendemos resolver estão em universos mais ricos em abstrações do que esse dos valores elementares das linguagens. Para descrever esses mundos, precisamos usar abstrações apropriadas para simplificar nosso discurso. Por exemplo, quando quero saber onde alguém mora, eu pergunto: Qual o seu ***endereço***? Quando quero saber quando um amigo nasceu, eu pergunto: Qual a ***data*** do seu nascimento? E quando quero saber para onde alguém vai deslocar o cavalo no jogo de xadrez, eu pergunto: Qual a nova ***posição***? Os nomes *endereço*, *data* e *posição* designam valores estruturados. Uma data tem três partes: dia, mês e ano. Um endereço pode ter quatro: rua, número, complemento e bairro. Já a posição no tabuleiro tem duas partes, linha e coluna.

Para possibilitar a descrição de valores dessa natureza, a linguagem Python dispõe de um construtor denominado ***tupla***. Podemos definir uma tupla como um agregado de dados, que possui quantidade pré-estabelecida de componentes, e onde cada componente da tupla pode ser de um tipo diferente (primitivo ou não).

11.2. Definição

A representação de uma tupla é feita com a seguinte sintaxe:

```
(t1, t2, t3, ..., tn)
```

onde cada ***ti*** é um termo da tupla. Se ***Ti*** é o tipo de ***ti***, então o universo de uma tupla é dado por:

```
TT = T1 x T2 x ... x Tn
```

Sabendo-se que os conjuntos bases dos tipos ***Ti*** são ordenados, também os elementos de ***TT*** o serão. Exemplos de tuplas:

	Intenção	Representação
1	uma data	`(15, 05, 2000)`
2	uma posição no tabuleiro de xadrez	`(3, 8)`
3	uma pessoa	`("Maria Aparecida", "solteira",(3, 02, 1990))`
4	um ponto no espaço	`(3.0, 5.2, 34.5)`
5	uma carta de baralho	`(7, "espada")`
6	uma tupla formada por expressões	`(x != 0 and y != 0,  x < 3 and y*x > 0)`
7	uma tupla formada por expressões	`(x, y, x + y)`

11.3. Compondo Tuplas

Os exemplos apresentados já parecem suficientes para que tenhamos entendido como descrever um valor do tipo tupla. Vamos então apenas comentar sobre os exemplos e ilustrar o uso de tuplas nas definições.

Uma tupla é um valor composto. Isto significa que devemos colocar entre parêntesis e separados por vírgulas os componentes deste valor. Cada componente é um valor, descrito diretamente ou através de expressões envolvendo operadores. Nos exemplos de 1 a 7, todos eles usam constantes. O exemplo 7 apresenta um dos valores descrito por uma expressão. No exemplo 3, um dos termos é uma outra tupla. Qualquer termo pode ser uma tupla. Um valor pode também ser paramétrico, como no exemplo 9, onde temos uma tupla de 3 termos, todos eles paramétricos, mas o terceiro depende dos dois primeiros. Podemos dizer que esta tupla descreve todas as triplas onde o terceiro termo pode ser obtido pela soma dos dois primeiros. No exemplo 8 podemos observar o uso de expressões condicionais. Observa-se aí também a descrição de um termo do tipo boolean, através de uma expressão relacional combinada com operadores lógicos. Vejamos agora alguns exemplos de uso de tuplas na descrição de valores.

Exemplo 1: Desejamos definir uma função, para que dados 2 valores, seja produzido uma tripla onde os dois primeiros termos são idênticos aos elementos fornecidos, em ordem inversa, e o terceiro termo seja igual à soma dos dois.

```
>>> def triplaSoma(a, b): return (a, b, a + b)
>>> triplaSoma(1,4)
(1, 4, 5)
```

Exemplo 2: Vamos definir uma função que produza o quociente e o resto da divisão inteira de dois números.

```
>>> def divisaoInt(x, y): return (x//y, x%y)
>>> divisaoInt(10,3)
(3, 1)
```

Exemplo 3: Voltemos à definição das raízes de uma equação do 2o. grau. Vimos anteriormente que, como eram duas, precisávamos definir também duas funções. Agora podemos agrupá-las em uma única definição:

```
>>> from math import sqrt
>>> def raizesEq2ograu(a, b, c):
        def x1(): return (-b + sqrt(delta()))/(2*a)
        def x2(): return (-b - sqrt(delta()))/(2*a)
        def delta(): return b*b - 4*a*c
        if a == 0: print("Funcao não definida!")
        elif delta() < 0: print("Nao ha raizes reais!")
        else: return (x1(), x2())
>>> raizesEq2ograu(1,0,-1)
(1.0, -1.0)
>>> raizesEq2ograu(1,0,-9)
(3.0, -3.0)
```

Exemplo 4: Voltemos aos movimentos do cavalo no jogo de xadrez. Vamos definir uma função que produza a nova posição, usando o primeiro movimento válido segundo o que se discutiu em **Resolvendo Problemas - Os Movimentos do Cavalo.**

```
>>> def movimentaCavalo(x, y):
        def pertTabuleiroXadrez((x, y)):
              return 1 <= x <= 8 and 1 <= y <= 8
        def m1(): return (x + 2, y - 1)
        def m2(): return (x + 1, y - 2)
        def m3(): return (x - 1, y - 2)
        def m4(): return (x - 2, y - 1)
        def m5(): return (x - 2, y + 1)
        def m6(): return (x - 1, y + 2)
        def m7(): return (x + 1, y + 2)
        def m8(): return (x + 2, y + 1)
        if pertTabuleiroXadrez(m1()): return m1()
        elif pertTabuleiroXadrez(m2()): return m2()
        elif pertTabuleiroXadrez(m3()): return m3()
        elif pertTabuleiroXadrez(m4()): return m4()
        elif pertTabuleiroXadrez(m5()): return m5()
        elif pertTabuleiroXadrez(m6()): return m6()
        elif pertTabuleiroXadrez(m7()): return m7()
        else: return m8()
>>> movimentaCavalo(4,5)
(6, 4)
>>> movimentaCavalo(5,4)
(7, 3)
```

Qual o valor de **`movimentaCavalo(1,9)`**? O que há de errado? Reescreva **`movimentaCavalo`** de forma a contornar o problema encontrado.

11.4. Selecionando termos de uma Tupla

Assim com precisamos compor uma tupla na descrição dos mapeamentos de uma função, também precisamos decompô-la. Quando uma função possui tuplas como parâmetros, para usar seus termos é necessário que se possa referenciá-los.

Exemplo: Desejamos determinar a distância entre dois pontos no plano.

Uma primeira solução é usar funções seletoras como **`prim`** e **`seg`** para extrair os elementos da tupla:

```
>>> def distancia(p1, p2):
        def prim((x,y)): return x
        def seg((x,y)): return y
        def dx(): return prim(p1) - prim(p2)
        def dy(): return seg(p1) - seg(p2)
        return sqrt(dx()**2 + dy()**2)
>>> distancia((0,0),(3,4))
5.0
```

Diversamente, podemos também nomear os elementos internos das tuplas na interface da função:

```
>>> def distancia((x1,y1), (x2, y2)):
```

```
        def dx(): return x1 - x2
        def dy(): return y1 - y2
        return sqrt(dx()**2 + dy()**2)
>>> distancia((0,0),(3,4))
5.0
```

EXERCÍCIOS:

1. Escreva uma função que determina as distâncias entre três pontos no plano cartesiano, duas a duas. Tanto os pontos dados como entrada, como as distâncias calculadas devem ser representadas por tuplas. Na resposta, cada par de pontos deve ser exibido, seguido da distância existente entre eles.
2. Dados a sua data de nascimento e a data atual, informe qual é a sua idade.

12. LISTAS

12.1. Introdução

A maioria dos itens de informação de nosso interesse está agrupada, dando origem a um conceito muito importante para a resolução de problemas, o agrupamento. Frequentemente estamos interessados em manipular esses agrupamentos para extrair informações, definir novos itens ou avaliar propriedades desses agrupamentos.

Tratamos anteriormente das tuplas, que são agrupamentos de tamanho predefinido e heterogêneo, ou seja, cada elemento que participa do agrupamento pode ser de um tipo diferente. Agora estamos interessados em explorar outro tipo de agregação, as listas. Este novo tipo, em Python, caracteriza-se por agregar quantidades variáveis de elementos desde que todos eles sejam de um mesmo tipo.

Vivemos cercados de listas. Elas estão em qualquer lugar onde precisamos registrar e processar dados. Vejamos alguns exemplos:

1. Lista de números pares;
2. Lista dos livros lidos por uma pessoa;
3. Lista dos amigos que aniversariam em um dado mês;
4. Lista dos presidentes corruptos;
5. Lista dos vereadores decentes;
6. Lista das farmácias enganadoras;
7. Lista das disciplinas que já cursei;
8. Lista dos lugares que visitei;
9. Lista dos números feios;
10. Lista dos números primos;
11. Lista das posições para as quais um cavalo pode se deslocar;
12. Lista das palavras de um texto;
13. Lista dos erros provocados pelo Windows;
14. Lista dos prêmios Nobel ganhos pelo Bertrand Russel;
15. Lista dos títulos conquistados pelo Nacional Futebol Clube;
16. Listas dos *funks* compostos pelo Noel Rosa.

Destas, algumas são vazias, outras são finitas e algumas infinitas.

12.2. Conceitos básicos:

Uma lista é uma sequência de zero ou mais elementos de um mesmo tipo. Entende-se por sequência uma quantidade qualquer de itens dispostos linearmente. Podemos representar uma lista pela enumeração dos seus elementos, separados por vírgulas e cercados por colchetes.

[*e1*, *e2*, ..., *en*]

Por exemplo:

```
>>> [ ]
>>> [1,2,3,4,5]
```

```
>>> ['a', 'e', 'i', 'o', 'u']
>>> [(22,04,1500), (07,09,1822), (31,03,1964)]
>>> [[1,2,5,10], [1,11], [1,2,3,4,6,12], [1,13], [1,2,7,14], [1,3,5,15]]
```

É importante ressaltar que em uma lista podemos falar do primeiro elemento, do quinto, do enésimo, ou do último. Ou seja, há uma correspondência direta entre os números naturais e a posição dos elementos de uma lista.

Este último fato nos lembra de um equívoco frequente, que queremos esclarecer de saída. A ordem que se adota em listas, por ser baseada nos números naturais, começa do zero. Ou seja, o primeiro elemento de uma lista tem o número de ordem igual a zero.

Por exemplo, na relação acima apresentada, a primeira lista é vazia. Na lista do item 4 o elemento de ordem 1 é a tupla (07, 09,1822) e na lista do item 5, o elemento de ordem zero (0) é a lista [1,2,5,10].

Quanto ao tipo, podemos dizer que a segunda lista é uma lista de números, a terceira uma lista de caracteres, a quarta é uma lista de triplas de números e a quinta é uma lista de listas de números. Qual será o tipo da primeira lista?

Uma lista vazia é de natureza polimórfica, isto é, seu tipo depende do contexto em que seja utilizada, como veremos em momento oportuno.

12.3. Formas Alternativas para Definição de Listas

Além da forma básica, acima apresentada, também conhecida como enumeração, onde explicitamos todos os elementos, dispomos ainda das seguintes maneiras: definição por intervalo, definição por progressão aritmética e definição por compreensão. As duas primeiras formas são apresentadas a seguir e a terceira, posteriormente.

Definição por Intervalo

De uma forma geral, podemos definir uma lista explicitando os limites inferior e superior de um conjunto conhecido, que possua uma relação de ordem entre os elementos, no seguinte formato:

range(<limite inferior>**,**<limite superior>+1**)**

range(<num_de_elementos>**)** {inicio em 0}

Abaixo listamos algumas definições de listas definidas por intervalo:

```
>>> range(1,6)
[1, 2, 3, 4, 5]
>>> range(-2,3)
[-2, -1, 0, 1, 2]
>>> range(10,3)
[]
>>> range(3)
[0, 1, 2]
```

A linguagem Python permite apenas o uso da função range para números inteiros, retornando também uma lista de inteiros. Observe também que o intervalo é aberto no

limite superior, ou seja, este não é incluído na lista. Particulamente, quando o limite inferior é maior que o superior retorna-se uma lista vazia (**`range(10,3)`**).

Definição por Progressão Aritmética

Analogamente à definição anterior, podemos definir uma lista explicitando os limites inferior e superior e a razão de progressão aritmética utilizando a seguinte notação:

range(<limite inferior>**,**<limite superior>,<razão>**)**

Abaixo listamos algumas definições de listas definidas por intervalo em PA:

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

Observe que a notação anterior é uma simplificação da definição de lista por intervalo em PA de razão 1, omitindo-se a razão. Ademais, o exemplo anterior em que o intervalo com limite inferior maior que o superior retornava uma lista vazia (**`range(10,3)`**), dada uma razão negativa, retorna agora uma lista decrescente.

12.4. Operações Básicas

As listas, como já dissemos, são elementos da linguagem que podemos utilizar para descrever valores estruturados. Como nos demais tipos da linguagem, valores descritos por listas podem e devem ser usados na descrição de novos valores através da utilização de operações definidas sobre eles. A seguir são apresentadas algumas dessas operações.

in : avalia se um determinado elemento é membro de uma lista.

<elemento> **in** <lista>

```
>>> 7 in range(1,8)
True
>>> 3 in range(0,10,2)
False
```

len: descreve o tamanho de uma lista.

len(<lista>**)**

```
>>> len([])
0
>>> len(range(1,234,3))
78
```

indexação: podemos acessar cada termo de uma lista indicando sua a posição dentro da lista, considerando que o primeiro elemento tem a ordem zero.

<lista> **[**<índice>**]**

Em particular, o primeiro (conhecido como **head**) e o último elementos da lista são obtidos como:

<lista> **[0]** e <lista> **[**<tamanho da lista>-1**]**

```
>>> [1,2,3][0]
1
>>> [0,1,2,3][4-1]
3
>>> range(0,20,4)[3]
12
```

sub-listas: podemos descrever sublistas a partir de uma lista. Para tanto basta indicarmos a posição inicial e final da sub-lista, separadas pelo símbolo "**:**". A omissão de uma ou outra posição indica início ou final da lista respectivamente.

<lista> **[**<índice inicial>**:**<índice final> +1**]**

Em particular, as sublistas contendo todos os elementos menos o primeiro (conhecida como **tail**), e aquela contento todos menos o último são obtidas como:

<lista> **[1:]** e <lista> **[:**<tamanho da lista>-1**]**

```
>>> [0,1,2,3][:]
[0, 1, 2, 3]
>>> [0,1,2,3][:3]
[0, 1, 2]
>>> [0,1,2,3][1:]
[1, 2, 3]
>>> [0,1,2,3][:2]
[0, 1]
>>> [0,1,2,3][2:]
[2, 3]
>>> [0,1,2,3][1:3]
[1, 2]
```

concatenação: descreve uma nova lista obtida pela concatenação de uma lista de listas.

<lista> **+** <lista>

```
>>> [1,2,3] + [4,5]
[1, 2, 3, 4, 5]
>>> range(1,6) + range(4,0,-1)
[1, 2, 3, 4, 5, 4, 3, 2, 1]
```

Em particular, pode-se adicionar elementos em uma lista, no inicio, no final, ou ainda no meio conforme indicado a seguir:

[<elemento>**]** **+** <lista>

ou

<lista> **+** **[**<elemento>**]**

ou

<lista> **[:**<posição>**]** **+** **[**<elemento>**]** **+** <lista> **[**<posição>**:]**

```
>>> [0] + [1,2,3]
[0, 1, 2, 3]
>>> range(1,5) + [5]
[1, 2, 3, 4, 5]
>>> range(1,5)[:2] + [0] + range(1,5)[2:]
[1, 2, 0, 3, 4]
```

12.5. Definição por Compreensão

Uma lista pode ser descrita através da enumeração de seus elementos, como já vimos através de vários exemplos. Esta forma é também denominada de definição por extensão, visto que todos os componentes precisam ser explicitados.

Podemos também descrever listas através das condições que um elemento deve satisfazer para pertencer a ela. Em outras palavras, queremos descrever uma lista através de uma intenção. Esta forma é análoga à que já conhecemos para descrição de conjuntos. Por exemplo, é usual escrever a notação abaixo para descrever o conjunto formado pelo quadrado dos números naturais menores que 10.

P = {quadrado de x | x é natural e é menor que 10}

ou ainda mais formalmente,

P = { x^2 | x pertence a N e x <10}

Podemos observar, no lado direito da definição, que ela é formada por duas partes. A primeira é uma expressão que descreve os elementos, usando para isso termos variáveis que pertencem a um conjunto conhecido e satisfazem as condições estabelecidas na segunda parte.

expressão	x^2
variável	x
pertinência	x pertence a N
condição	x < 10

Em nosso caso, não estamos interessados em descrever conjuntos e sim listas. E isso tem algumas implicações práticas. Por exemplo, em um conjunto a ordem dos elementos é irrelevante, para listas não. É bom lembrar ainda que em uma lista, o mesmo valor pode ocorrer varias vezes, em diferentes posições.

A sintaxe que usaremos é a seguinte:

[<expressão> **for** <variável> **in** <lista-conhecida> **]**

ou

[<expressão> **for** <variável> **in** <lista-conhecida> **if** <condição>**]**

que define uma lista formada pela aplicação da expressão aos elementos da lista conhecida, desde que satisfazem a condição quando especificada.

Por exemplo, vejamos a descrição da lista dos quadrados dos números naturais menores que 5.

```
>>> [x**2 for x in range(0,5)]
[0, 1, 4, 9, 16]
```

O exemplo a seguir, descreve uma sublista de números ímpares, tendo como origem de geração uma lista definida por uma Progressão Aritmética.

```
>>> [x for x in range(1,100,4) if x%2==1]
[1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61, 65, 69, 73, 77,
81, 85, 89, 93, 97]
```

Vejamos exemplos de como usar este novo conceito na definição de funções.

Exemplo 1. Define uma função para gerar a sublista dos elementos pares de uma lista dada, e outra função que retorna uma lista formada pelos quadrados dos elementos pares de uma lista dada.

```
>>> def subListaPares(xs):
        def par(x): return x%2 == 0
        return [x for x in xs if par(x)]
>>> subListaPares(range(1,50,3))
[4, 10, 16, 22, 28, 34, 40, 46]

>>> def listaQuadPares(xs): return [x**2 for x in subListaPares(xs)]
>>> listaQuadPares(range(1,50,3))
[16, 100, 256, 484, 784, 1156, 1600, 2116]
```

Exemplo 2. Define uma função para gerar a sublista dos elementos de uma dada lista que são menores que um elemento fornecido

```
>>> subListaMenores(listaQuadPares(range(1,50,3)), 500)
[16, 100, 256, 484]
```

Exemplo 3. Define uma função que retorna uma lista formada pelos quadrados dos elementos de uma dada lista que são menores do que uma constante k, seguidos dos elementos maiores ou iguais a k acrescidos de duas unidades.

```
>>> def geraListaEmFuncaoDeK(xs, k):
        def f(x):
              if x < k: return x**2
              else: return x + 2
        return [f(x) for x in xs]
>>> geraListaEmFuncaoDeK(range(1,50,3),20)
[1, 16, 49, 100, 169, 256, 361, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51]
```

Podemos ainda usar mais de uma variável para percorrer listas diferentes, como no exemplo a seguir:

```
>>> [x*y for x in [0,1,2] for y in [5,10]]
[0, 0, 5, 10, 10, 20]
```

Finalmente, podemos também formar listas de listas ou de tuplas usando uma ou mais variáveis, como no exemplo a seguir:

```
>>> [(x, x**2) for x in range(1,9)]
[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64)]
>>> [[x, x**2] for x in range(1,9)]
[[1, 1], [2, 4], [3, 9], [4, 16], [5, 25], [6, 36], [7, 49], [8, 64]]
>>> [(x, y) for x in [1,2,3] for y in [-3,7,11]]
[(1, -3), (1, 7), (1, 11), (2, -3), (2, 7), (2, 11), (3, -3), (3, 7), (3,
11)]
```

Exemplo 4. Define uma função (constante) que gera uma lista de pontos do plano cartesiano, localizados no primeiro quadrante e delimitado pelas ordenadas 3 e 5.

```
>>> def pontos(): return [(x,y) for x in range(0,4) for y in range(0,6)]
>>> pontos()
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 0), (1, 1), (1, 2), (1,
3), (1, 4), (1, 5), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 0),
(3, 1), (3, 2), (3, 3), (3, 4), (3, 5)]
```

Ou ainda, usando condições:

```
>>> def pontos(): return [(x,y) for x in range(0,4) for y in range(0,6) if
x%2 == 0 and y%2 ==1]

>>> pontos()
[(0, 1), (0, 3), (0, 5), (2, 1), (2, 3), (2, 5)]
```

12.5. Outras Operações para manipulação de listas

> map(<função>, <lista-conhecida>) => <lista-resultante>

Esta operação consiste em aplicar uma dada função a cada elemento de uma lista conhecida, produzindo-se então uma lista resultante com exatamente a mesma quantidade de elementos. Observe, porém, que o tipo de dado da lista resultante pode diferir da lista original, pois é determinado pelo tipo de retorno da função. A função, por sua vez, deve ter apenas um parêmetro, compatível com os elementos da lista original.

```
>>> def f(x): return 2*x
>>> map(f, range(1,6))
[2, 4, 6, 8, 10]

>>> def f(x): return x%2==0
>>> map(f, range(1,6))
[False, True, False, True, False]
```

Como seriam as construções equivalentes aos exemplos dados usando definição por *comprehension* (abrangência/compreensão)?

> filter(<função>, <lista-conhecida>) => <lista-resultante>

Esta operação consiste em aplicar um filtro a uma lista conhecida. Este filtro é dado por uma função de verificação, de forma que a lista resultante terá como elementos apenas

aqueles para os quais a avaliação da função é verdadeira. Observe que a lista resultante será sempre uma sublista da lista original, e terá uma quantidade de elementos menor ou igual. A função, portanto, deve ser de verificação e ter apenas um parêmetro, compatível com os elementos da lista original.

```
>>> def f(x): return x%2==0
>>> filter(f, range(1,6))
[2,4]
```

Como seria uma construção equivalente ao exemplo dado usando definição por *comprehension* (abrangência/compreensão)?

> reduce(<função>, <lista-conhecida>) => <valor-resultante>

> reduce(<função>, <lista-conhecida>, <valor-inicial>) => <valor-resultante>

Esta operação consiste em aplicar de forma acumulativa/composta uma dada função aos elementos de uma lista conhecida. A função precisa ter exatamente dois parâmetros compatíveis com o tipo dos elementos da lista, e seu resultado deve ser também compatível pois será usado na reaplicação da função aos elementos seguintes.

```
>>> def f(x,y): return x+y
>>> reduce(f, range(1,6))            #f( f( f( f(1,2), 3),  4),  5 )
15
>>> reduce(f, range(1,6),50)         #f( f( f( f( f( 50,1) ,2), 3),  4),  5 )
65
```

> lambda p1, p2, ..., pn: expressão

Uma expressão lambda permite escrever funções anônimas, e, portanto, economizar na definição de funções simples:

```
>>> map(lambda x: 2*x, range(1,6))
[2, 4, 6, 8, 10]
>>> filter(lambda x: x%2==0, range(1,6))
[2, 4]
>>> reduce(lambda x, y: x+y, range(1,6))
15
```

> zip(lista1, lista2, ..., lista-n) -> lista-enuplas

Esta operação consiste produzir uma lista com enuplas formadas pelos elementos de mesma ordem de cada uma das n listas informadas como parâmetro. Observe que a quantidade de elementos de cada tupla é igual à quantidade de listas, e a quantidade de tuplas é igual ao tamanho da menor lista informada.

```
>>> zip(range(1,6), range(2,7), range(3,8))
[(1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 6), (5, 6, 7)]
>>> zip([1,2,3,4,5],[8,7,6])
[(1, 8), (2, 7), (3, 6)]
```

Como seriam as construções equivalentes aos exemplos dados usando definição por *comprehension* (abrangência/compreensão)?

EXERCÍCIOS:

1. Escreva uma função que determina as distâncias entre três pontos no plano cartesiano, duas a duas. Tanto os pontos dados como entrada, como as distâncias calculadas devem ser representadas por tuplas. Na resposta, cada par de pontos deve ser exibido, seguido da distância existente entre eles.
2. Dados a sua data de nascimento e a data atual, informe qual é a sua idade.
3. Defina uma função que retorne uma sub-lista com os k primeiros elementos de uma dada lista. f(3, range(0,10)) => [0,1,2]
4. Defina uma função que retorne uma sub-lista com os elementos a partir dos k primeiros de uma dada lista. g(3, range(0,10)) => [3,4,5,6,7,8,9]

13. RESOLVENDO PROBLEMAS COM LISTAS

Neste capítulo desenvolveremos a solução para alguns problemas mais complexos. A intenção é apresentar o uso de estratégias de propósito geral que podem ser úteis na resolução de novos problemas. Discutimos também algumas questões relativas ao desempenho das soluções.

13.1 PROBLEMA 1: *Dada uma lista, determine o seu maior elemento.*

Comecemos por definir, usando a linguagem de conjuntos, quem é este elemento.

Dizemos que k é o maior elemento de um conjunto C, se e somente se, o subconjunto de C formado por todos os elementos maiores que k é vazio.

Em linguagem de lista isto equivale a dizer que se C é uma lista, a sublista de C formada pelos caras de C maiores que k é vazia.

Vejamos como fica a codificação em Python.

```
>>> def maiores(k, lista):
        """
        Funcao para determinar os elementos maiores que um dados x
        em uma lista
        """
        return [x for x in lista if x > k]

>>> maiores(3, [1,4,3,6,8,7,8])
[4, 6, 8, 7]

>>> def maiorais(lista):
        """
        Funcao para determinar os maiores elementos em uma lista,
        considerando-se que o maior numero pode ter varios exemplares
        """
        return [x for x in lista if maiores(x, lista) == []]

>>> maiorais([1,4,3,6,8,7,8])
[8, 8]

>>> def maximo(lista):
        """
        Funcao para determinar o maior elemento de uma lista,
        considerando-se que o maior numero pode ter varios exemplares
        """
        if lista == []: print ('Lista Vazia')
        else: return maiorais[0]

>>> maximo([1,4,3,6,8,7,8])
8
```

13.2 PROBLEMA 2: *Dada uma lista, verifique se ela é não decrescente.*

Como aquecimento, vamos considerar listas de números e a noção usual de ordem não decrescente. Antes de programar, vamos resgatar a definição de sequências não decrescentes.

Definição: Uma sequência S está em ordem não decrescente se e somente se qualquer um de seus elementos é menor ou igual aos seus sucessores. Em outras palavras, podemos dizer que a coleção de elementos de S que são maiores que seus sucessores é vazia.

```
>>> def maioresQueSucessores(lista):
    """
    Funcao para determinar os elementos de uma lista,
    que sao maiores que algum dos seus sucessores
    """
    return [ lista[i] for i in range(0,len(lista)-1) for j in
range(i+1,len(lista)) if lista[i] > lista[j]]

>>> maioresQueSucessores([1,4,5,3,6,8,7,8])
[4, 5, 8]

>>> def estahEmOrdemNaoDecrescente(lista):
    """
    Funcao para determinar se os elementos de uma lista estao em ordem não
    decrescente
    """
    return maioresQueSucessores(lista) == []

>>> estahEmOrdemNaoDecrescente([1,4,5,3,6,8,7,8])
False

>>> estahEmOrdemNaoDecrescente([1,3,6,7,8])
True
```

13.3. Avaliando a solução encontrada:

Podemos avaliar a da solução encontrada em duas dimensões: exploração adequada das propriedades do problema e escolha adequada dos mecanismos da linguagem. A seguir fazemos uma pequena exploração desses dois aspectos.

13.3.1. Explorando propriedades do problema - Analisando a nossa definição anterior constatamos que ela diz mais do que precisamos. Ela avalia cada elemento com respeito a todos sucessores. Na verdade, nossa definição pode ser melhorada. Basta saber que cada elemento tem que ser menor ou igual ao seu sucessor imediato. Vamos reescrever a nossa definição:

Definição : Uma sequência S está em ordem não decrescente se e somente se qualquer um de seus elementos é menor ou igual ao seu **sucessor imediato**. Em outras palavras, podemos dizer que a coleção de elementos de S que são maiores que seus **sucessores imediatos** é vazia.

Vejamos então a nova implementação e sua aplicação às mesmas instâncias do problema:

```
>>> def maioresQueSucessorImediato(lista):
     """
     Funcao para determinar os elementos de uma lista
     que sao maiores seu sucessor imediato
     """
     return [lista[i] for i in range(0,len(lista)-1) if lista[i] > lista[i+1]]

>>> maioresQueSucessorImediato([1,4,5,3,6,8,7,8])
[5, 8]
```

13.3.2. Explorando os mecanismos da linguagem - Outra investida que podemos fazer é com respeito ao uso adequado dos mecanismos da linguagem. As soluções acima apresentadas processam as listas através de índices, ou seja, fazem um acesso aleatório aos elementos da lista. Porém, o acesso sequencial possui realização mais eficiente.

```
>>> def adjacentes(lista):
     """
     Funcao que retorna uma lista de tuplas formada por cada elemento da lista
     dada e seu sucessor imediado. Aplica-se a função zip da lista original com
     a sublista iniciada do segundo elemento.
     """
     return zip (lista, lista[1:])

>>> adjacentes([1,4,5,3,6,8,7,8])
[(1, 4), (4, 5), (5, 3), (3, 6), (6, 8), (8, 7), (7, 8)]

>>> def maioresQueSucessorImediato(lista):
     """
     Funcao para determinar os elementos de uma lista
     que sao maiores seu sucessor imediato
     """
     return [x for (x,y) in adjacentes(lista) if x > y]

>>> maioresQueSucessorImediato([1,4,5,3,6,8,7,8])
[5, 8]
```

EXERCÍCIOS:

1. Dada uma lista l, contendo uma quantidade igual de números inteiros pares e ímpares (em qualquer ordem), defina uma função que, quando avaliada, produz uma lista na qual esses números pares e ímpares encontram-se alternados.
2. Dada uma lista xs, fornecer uma dupla contendo o menor e o maior elemento dessa lista.
3. Dadas duas listas de elementos distintos, determinar a união delas.

14. PARADIGMA APLICATIVO

A descrição de funções, como acontece com outras formas de representação de conhecimento, admite vários estilos. Dependendo do problema que estamos tratando, um estilo pode ser mais conveniente que outros. Podemos dizer que influi muito na escolha, o quanto desejamos ou necessitamos, falar sobre como computar uma solução. A descrição de soluções tem um espectro de operacionalidade que vai do declarativo até o procedural, em outras palavras, do **que** desejamos computar ao **como** queremos computar.

Uma situação que ocorre com frequência na descrição de funções é a necessidade de aplicar uma função, de forma cumulativa, a uma coleção de elementos. Em outras palavras, desejamos generalizar uma operação para que seu uso se estenda a todos os elementos de uma lista. Chamemos este estilo de paradigma aplicativo.

Por exemplo, suponha que desejamos obter a soma de todos os elementos de uma lista. A operação ***adição* (+)** segundo sabemos, é de natureza binária, ou seja, opera sobre dois elementos produzindo um terceiro. Para operar sobre todos os elementos de uma lista de forma a obter a soma total, podemos operá-los dois a dois, obtendo com isso resultados intermediários, que por sua vez poderão ser novamente operados e assim sucessivamente até que se obtenha o resultado final. Observemos o processo para uma instância:

Obter a soma dos elementos da lista [5, 9, 3, 8, 15, 16]

Podemos tomar os pares de elementos, primeiro com segundo, terceiro com quarto e assim sucessivamente.

expressão	nova expressão	redução
+ [5, 9, 3, 8, 15, 16]	+ [14, 3, 8, 15, 16]	5 + 9 = 14
+ [14, 3, 8, 15, 16]	+ [14, 11, 15, 16]	3 + 8 = 11
+ [14, 11, 15, 16]	+ [14, 11, 31]	15 + 16 = 31
+ [14, 11, 31]	+ [25, 31]	14 + 11 = 25
+ [25, 31]	+ [56]	25 + 31 = 56
+ [56]	56	

A escolha dos pares, no caso da adição, não importa. As reduções poderiam ser aplicadas em qualquer ordem, visto que a operação é adição é comutativa. Se, pode ser qualquer uma, então podemos estabelecer uma ordem, como por exemplo, da esquerda para a direita, usando em cada nova redução o elemento resultante da redução anterior.

Expressão	nova expressão	Redução
+ [5, 9, 3, 8, 15, 16]	+ [14, 3, 8, 15, 16]	5 + 9 = 14
+ [14, 3, 8, 15, 16]	+ [17, 8, 15, 16]	14 + 3 = 17
+ [17, 8, 15, 16]	+ [25, 15, 16]	17 + 8 = 25
+ [25, 15, 16]	+ [40, 16]	25 + 15 = 40
+ [40, 16]	+ [56]	40 + 16 = 56
+ [56]	56	

Em *Python* o operador **reduce** é que permite a descrição de computações desta natureza. Como vimos, ele tem como parâmetros a **função** a ser aplicada aos elementos da lista cumulativamente, a **lista** a ser manipulada, e, opcionalemente, um ***valor inicial*** para a operação considerada. Este elemento é tomado como ponto de partida para as reduções, ou seja, a primeira aplicação da operação é sobre o ***valor inicial*** e o primeiro elemento da lista. Eis a sintaxe:

reduce (<função>, <lista>) ou **reduce** (<função>, <lista>, <valor inicial>)

A ordem estabelecida para as reduções é semelhante à ilustração acima, ou seja, caminha-se da esquerda para direita, usando o resultado da redução anterior para a nova redução.

Vejamos a definição da função ***soma*** cujo objetivo é a descrição da soma dos elementos de uma lista.

```
>>> def soma(lista):
        def f(x, y): return x + y
        return reduce(f, lista, 0)

>>> soma([1,2,3,4,5])          #f( f( f( f( f( 0,1) ,2), 3),  4),  5 )
15
```

A função **`f`** é aplicada aos elementos da função na forma f(... f (f(0, x0), x1, ...) , xn), ou seja, o resultado de f(0, x0) é usado como parâmetro na aplicação da função ao elemento x1, e este resultado é usado na aplicação da função ao elemento x2, e assim por diante até o último elemento da lista. Se o elemento inicial não é informado, a primeira aplicação da função ocorre entre x0 e x1, ou seja, f(x0, x1).

A indicação do valor inicial é importante para determinar o resultado no caso em que a lista informada é vazia. Se ele não for informado, este caso resultará em erro. Outra possibilidade seria incluir uma condição de proteção antes da aplicação do reduce. Em exemplos futuros veremos outros usos para o valor especial.

14.1. ALGUMAS OPERAÇÕES IMPORTANTES

Assim como a somatória, existem outras operações importantes, de grande utilidade, que podem ser obtidas pelas seguintes equações usando ***reduce.***

```
>>> def produto(lista):
        def f(x, y): return x * y
        return reduce(f, lista, 1)

>>> def conjuncao(lista):
        def f(x, y): return x and y
        return reduce(f, lista, True)

>>> def disjuncao(lista):
        def f(x, y): return x or y
        return reduce(f, lista, False)
```

Exemplo 01: Usando **produto** para descrever o fatorial de um número. Sabemos que: *O fatorial de 0 é 1, e o de um número natural n>0 é igual ao produto de todos os números naturais de 1 até n.*

```
>>> def fatorial(n):
        return produto(range(1,n+1))

>>> fatorial(5)
120
>>> fatorial(0)
1
>>> fatorial(1)
1
```

Exemplo 02: Podemos usar a disjunção generalizada para definir uma função que avalia se em uma dada lista de números pelo menos um deles é impar.

```
>>> def aoMenosUmPar(lista):
        def par(x): return x%2==0
        return disjuncao(map(par, lista))

>>> aoMenosUmPar([1,3,5,7,9])
False
>>> aoMenosUmPar([1,3,4,7,9])
True
```

Ou ainda, podemos aplicar o conceito de disjuncao diretamente com o paradigma aplicativo.

```
>>> def aoMenosUmPar(lista):
        def f(x,y): return x or y%2==0
        return reduce(f, lista, False)

>>> aoMenosUmPar([1,3,5,7,9])
False
>>> aoMenosUmPar([1,3,4,7,9])
True
```

14.2. O MENOR ELEMENTO DE UMA LISTA

Estamos interessados em obter uma função que associe uma lista com menor de todos os seus elementos. Anteriormente já apresentamos uma versão para a função que descreve o maior elemento de uma lista, que é bastante similar a esta. Na oportunidade exploramos uma propriedade que o elemento ***maior*** de todos deve satisfazer. Poderíamos explorar a propriedade do menor elemento de forma análoga.

Em uma lista xs, dizemos que k é o menor elemento de xs, se e somente se a sublista de xs formada por elementos menores que k é vazia.

Porém, vamos explorar agora o problema a partir da generalização da operação **menor.** Em sua forma básica, a função **menor** associa dois números quaisquer com o menor entre eles. Assim, a operação **reduce** aplica a função **menor** aos dois primeiros elementos, e ao resultado é aplicado novamente a função junto ao terceiro elemento, e assim por diante até

o último elemento da lista. Desta forma o resultado final será justamente o menor elemento dentre todos.

```
>>> def minimo(lista):
        def menor(x, y):
              if x < y: return x
              else: return y
        if lista == []: print('Funcao nao definida: lista vazia!')
        else: return reduce(menor, lista)

>>> minimo([3,10,5,2,7,9])
2
```

14.3. Inversão de uma lista

Dada uma lista xs, desejamos descrever a lista formada pelos elementos de xs tomados em ordem inversa. Para resolver o problema precisamos inventar uma função básica passível de generalização.

Vamos começar descrevendo a função **insereAntes.**

```
>>> def insereAntes(lista, x):
        return [x] + lista

>>> insereAntes([2,3,4,5],1)
[1, 2, 3, 4, 5]
```

Tentemos agora a generalização. A intenção é incluir cada elemento da lista xs que desejamos inverter, antes do primeiro elemento de uma lista que iremos construindo gradativamente. O ***valor inicial*** será a lista vazia. Vejamos um exemplo onde inverteremos a lista [0, 1, 2, 3, 4]

lista parcial	novo elemento	redução
[]	0	[0]
[0]	1	[1, 0]
[1, 0]	2	[2, 1, 0]
[2, 1, 0]	3	[3, 2, 1, 0]
[3, 2, 1, 0]	4	[4, 3, 2, 1, 0]

Vamos então usar o operador **reduce** para construir a generalização desejada:

```
>>> def inverteLista(lista):
        return reduce(insereAntes, lista, [])

>>> inverteLista([3,4,5,6])
[6, 5, 4, 3]
```

14.4. Inserção ordenada e ordenação de uma lista

A ordenação de dados é uma das operações mais realizadas em computação. Em todos os computadores do mundo, faz-se uso intensivo dela. Este assunto é muito especial e por isso mesmo profundamente estudado. Cabe-nos aqui fazer uma breve passagem pelo

assunto, sem, contudo nos aprofundarmos nas questões de eficiência, que é central no seu estudo.

Será que podemos usar o conceito de generalização de uma operação para descrever a ordenação de uma lista? Que operação seria essa?

14.4.1. Inserção em lista ordenada - Vamos começar discutindo outra questão mais simples: dada uma lista, com seus elementos já dispostos em ordem não decrescente, como descrever uma lista na mesma ordem, acrescida de um elemento também fornecido?

Podemos observar que se a lista xs está em ordem não decrescente, com respeito ao elemento x (dado), podemos descrevê-la através de dois segmentos:

xs = <menores que x> + <maiores ou iguais a x>

Para acrescentar x a xs, basta concatenar x entre os dois segmentos, obtendo a nova lista ys, assim:

ys = <menores que x em xs> + [x] + <maiores ou iguais a x em xs>

Por exemplo, se queremos inserir o número 3, na lista [0,1,2,4,5], usando o paradigma aplicativo, podemos tentar a seguinte solução:

i) Construir, a partir da lista vazia, a lista parcial dos elementos menores que 3

ii) Inserir o número 3 na lista parcial

iii) Completar a lista parcial com os elementos maiores ou iguais a 3

lista parcial	elemento da lista	verificação	redução
[]	0	0 < 3	[] + [0]
[0]	1	1 < 3	[0] + [1]
[0, 1]	2	2 < 3	[0, 1] + [2]
[0, 1, 2]	4	4 >= 3	[0, 1, 2] + [3] + [4]
[0, 1, 2, 3, 4]	5	5 >= 3	[0, 1, 2, 3, 4] + [5]

Podemos vislumbrar, a partir dessa idéia, a seguinte solução inicial:

```
>>> def insereOrdem(lista, x):
        def f(ls, elem):
              if elem < x: return ls + [elem]
              else: return ls + [x] + [elem]
        if lista == []: return [x]
        else: return reduce(f, lista, [])
```

Observe, porém, que esta solução apresenta alguns problemas. Vejamos as instanciações a seguir:

```
>>> insereOrdem([1,2,4], 3)
[1, 2, 3, 4]                          # OK!
>>> insereOrdem([1,2,4], 0)
[0, 1, 0, 2, 0, 4]                    # Ops, inserimos o 0 três vezes!
>>> insereOrdem([1,2,4], 5)
[1, 2, 4]                             # Ops, o 5 não foi inserido
>>> insereOrdem([1,2,4,5], 3)
[1, 2, 3, 4, 3, 5]                    # Ops, inserimos o 3 duas vezes!
```

Precisamos, então, melhorar a nossa solução incial para resolver os problemas encontrados. Primeiramente, podemos tratar em particular os casos em que o número a ser inserido é menor ou igual ao primeiro elemento da lista, ou maior ou igual ao último. Para estes casos, a inserção ordenada pode adicionar o elemento ao início da lista no primeiro caso, ou adicioná-lo ao final da lista no segundo caso. Essa solução, implementada a seguir, resolve os dois primeiros problemas detectados acima.

```
>>> def insereOrdem(lista, x):
        def f(ls, elem):
              if elem < x: return ls + [elem]
              else: return ls + [x] + [elem]
        if lista == []: return [x]
        elif x <= lista[0]: return [x] + lista
        elif x >= lista[len(lista)-1]: return lista + [x]
        else: return reduce(f, lista, [])
```

Agora nos falta evitar a re-inserção do número na lista após termos encontrado o primeiro elemento maior ou igual a ele. Assim, nos interessa verificar se o número já foi inserido.

lista parcial	numInserido	elemento da lista	verificação	redução
[]	False	0	0 < 3	[] + [0]
[0]	False	1	1 < 3	[0] + [1]
[0, 1]	False	2	2 < 3	[0, 1] + [2]
[0, 1, 2]	False	4	4 >= 3	[0, 1, 2] + [3] + [4]
[0, 1, 2, 3, 4]	True	5	5 >= 3	[0, 1, 2, 3, 4] + [5]

Para isso, vamos acrescentar à função que vai ser aplicada à lista um parâmetro que contém a verificação da inserção do elemento. Lembre-se de que este parâmetro deve acompanhar a lista a ser construída usando-se uma tupla, pois a função a ser aplicada via **`reduce`** deve ter apenas 2 parâmetros, sendo o segundo deles é aplicado aos elementos da lista. Observe também que o valor inicial de tal parâmetro é **`Falso`**, e só virá a ser **`Verdadeiro`** quando o primeiro elemento maior ou igual o número for encontrado. Neste caso, o número será inserido na lista parcial antes do elemento atual, e, a partir de então, a condição de verificação passa a ter valor **`Verdadeiro`**. Os demais elementos, maiores ou iguais ao número, serão também adicionados ao final da lista.

Uma implementação desta solução é dada pelo código:

```
>>> def insereOrdem(lista, num):
      def f((ls, numInserido), elem):
       if not numInserido and elem >= num: return (ls + [num] + [elem], True)
       else: return (ls + [elem], numInserido)
      def prim((x,y)): return x
      if lista == []: return [num]
      elif num <= lista[0]: return [num] + lista
      elif num >= lista[len(lista)-1]: return lista + [num]
      else: return prim(reduce(f, lista, ([],False)))

>>> def insereOrdem([1,2,4],3)
[1,2,3,4]
>>> insereOrdem([1,2,4,5], 3)
[1, 2, 3, 4, 5]
>>> insereOrdem([1,2,4], 0)
```

```
[0, 1, 2, 4]
>>> insereOrdem([1,2,4], 5)
[1, 2, 4, 5]
>>> insereOrdem([], 3)
[3]
```

14.4.2. Ordenação - A aplicação sucessiva de **insereOrdem**, conforme ilustrado abaixo, nos dá uma pista para nossa generalização. Podemos pegar cada elemento da lista a ser ordenada e inseri-lo em ordem em outra lista que será paulatinamente construída, já ordenada. Vamos verificar essa idéia na ordenação da lista [10,20,30,5,25]:

lista parcial	**novo elemento**	**redução**
[]	10	[10]
[10]	20	[10,20]
[10,20]	30	[10,20,30]
[10,20,30]	5	[5, 10, 20, 30]
[5, 10, 20, 30]	25	[5, 10, 20, 25, 30]

O ponto de partida, neste caso, é a lista vazia. Vamos tomar então a lista vazia como o valor inicial. Vejamos como fica então nossa definição para ordenação de listas.

```
>>> def ordenaLista(lista):
  return reduce(insereOrdem, lista, [])

>>> ordenaLista([10,20,5,15])
[5,10,15,20]

>>> ordenaLista([10])
[10]

>>> ordenaLista([2,-9,1])
[-9,1,2]
```

EXERCÍCIOS:

Use o paradigma aplicativo para resolver os problemas abaixo:

1. Dadas duas listas xs e ys, verificar se xs é sublista de ys.
2. Dada uma string (lista de caracteres), verificar se é um palíndrome (a string é a mesma quando lida da esquerda para a direita ou da direita para a esquerda, ex: “arara” – observe que a string vazia é dada por “” ao invés de [ ]).
3. Gere uma lista com a sequencia de fibonacci até um numero n.

15. O PARADIGMA RECURSIVO

Como já falamos anteriormente, existem várias maneiras de definir um conceito. A essas maneiras convencionamos chamar de paradigmas. Aqui trataremos de mais um destes, o paradigma recursivo. Dizer que trataremos de mais um é simplificar as coisas, na verdade este paradigma é um dos mais ricos e importantes para a descrição de computações. O domínio deste paradigma é de fundamental importância para todo aquele que deseja ser um *expert* em Programação de Computadores enquanto ciência e tecnologia.

De uma maneira simplificada podemos dizer que o núcleo deste paradigma consiste em descrever um conceito de forma recursiva. Isto equivale a dizer que definiremos um conceito usando o próprio conceito. Apesar de disto parecer muito intrigante, não se assuste, aos poucos, quando esboçarmos melhor a idéia ela se mostrará precisa, simples e poderosa.

Vamos pensar num conceito bem corriqueiro. Que tal definirmos o conceito ***escada***. Como podemos descrever escada usando a própria escada? A resposta é bem simples:

Uma escada é igual a um degrau seguido de uma escada **(Figura 15.1).**

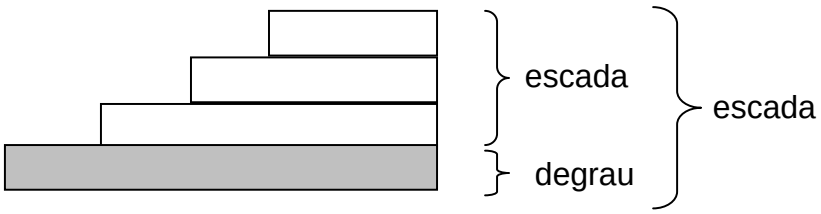


Figura 15.1 – Uma escada

Fácil não é? Será que isto basta? Onde está o truque? Parece que estamos andando em círculo, não é mesmo? Para entender melhor vamos discutir a seguir alguns elementos necessários para a utilização correta da recursão na definição de novas funções.

15.1. Descrição recursiva de um conceito familiar

Antes de avançar em nossa discussão vamos apresentar mais um exemplo. Desta vez usaremos um que é bastante familiar para alunos de ciências exatas. Estamos falando da descrição do fatorial de um número. Já vimos neste curso uma forma de descrever este conceito quando estudamos o paradigma aplicativo. Na oportunidade usamos a seguinte descrição:

*O fatorial de um número natural **n** > 0 é igual*
*ao produto de todos os números naturais de 1 até **n**.*

Ou ainda em notação mais formal:

$$n ! = 1 \times 2 \times 3 \times \ldots \times n$$

Em Python, como já vimos, teremos a seguinte definição:

```
>>> def fatorial(n):
        def prod(x, y): return x * y
        return reduce(prod, range(1,n+1), 1)
```

Há ainda outra forma de definir o fatorial de um número natural, também familiar aos alunos do primeiro ano universitário:

O Fatorial de um número natural n é igual ao produto deste número pelo fatorial de seu antecessor.

Novamente, sendo mais formal, podemos escrever:

$$n ! = n \times (n - 1) !$$

E em Python, como ficaria? Que tal a definição a seguir?

```
>>> def fatorial(n):
        return n * fatorial(n-1)
```

Bom, se tentarmos chamar a função acima para qualquer valor de n constataremos que há algum problema com nossa definição. Este problema é decorrente de da falta de um detalhe que faz toda a diferença. Na verdade a nossa definição recursiva para fatorial está incompleta: esta que exibimos é a definição geral para os naturais maiores que zero. A definição do fatorial de zero não é recursiva, ela é independente:

O fatorial de zero é igual a 1.

Temos então duas definições para fatorial e precisamos integrá-las. Vejamos uma tentativa:

O Fatorial de um número natural n é:

1. *igual a 1 se n=0;*
2. *igual ao produto deste número pelo fatorial de seu antecessor, se n > 0*

Vamos ver como essa integração pode ser feita em Python. Podemos de imediato observar que trata-se de uma definição condicional e logo nos vem a lembrança de que nossa linguagem possui um mecanismo, as expressões condicionais.

```
>>> def fatorial(n):
        if n == 0: return 1
        else return n * fatorial(n-1)

>>> fatorial(0)
1
>>> fatorial(1)
1
>>> fatorial(5)
120
```

15.2. Elementos de uma Descrição Recursiva

Em uma descrição recursiva devemos ter em conta certo elementos importantes. É fundamental que todos eles sejam contemplados para que nossas descrições estejam corretas. O exemplo anteriormente apresentado é suficiente para ilustrar todos eles. Vamos então discuti-los:

Definição geral: Toda definição recursiva tem duas partes, uma delas se aplica a um valor qualquer do domínio do problema, denominamos de geral. Esta tem uma característica muito importante, o conceito que está sendo definido deve ser utilizado. Por exemplo, para definir fatorial de n, usamos o fatorial do *antecessor de n*. Observe aqui, entretanto que o mesmo conceito foi utilizado, mas não para o mesmo valor. Aplicamos o conceito a um valor mais simples, neste caso o antecessor de n.

Definição independente: A outra parte da definição é destinada ao tratamento de um valor tão simples que a sua definição possa ser dada de forma independente. Este elemento é também conhecido como base da recursão. No caso do fatorial, o valor considerado é o zero.

Obtenção de valores mais simples: Para aplicar o conceito a um valor mais simples precisamos de uma função que faça este papel. No caso do fatorial, usamos a subtração de n por 1, obtendo assim o antecessor de n. Em cada caso, dependendo do domínio do problema e do problema em si, precisaremos encontrar a função apropriada.

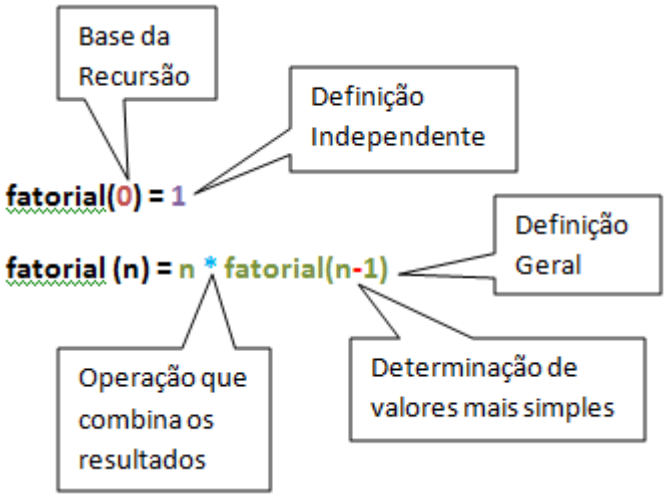
Função auxiliar: Na definição geral, para obter um valor usando o valor considerado e o valor definido recursivamente, em geral faz-se necessário o uso de uma função auxiliar. Algumas vezes esta função pode ser originada a partir de um conceito aplicável a dois elementos e que desejamos estender aos elementos de uma lista. Um exemplo é o caso da somatória dos elementos de uma lista, como veremos adiante. No caso do fatorial esta função é a multiplicação.

Garantia de atingir o valor independente: É fundamental que a aplicação sucessiva da função que obtém valores mais simples garanta a determinação do valor mais simples. Este valor é também denominado de base da recursão. Por exemplo, no caso do fatorial, sabemos que aplicando a subtração sucessivas vezes produziremos a sequência:

n, (n-1), (n-2), ... 0

Esta condição é fundamental para garantir que ao avaliarmos uma expressão atingiremos a base da recursão.

Voltemos à definição do fatorial para destacarmos os elementos acima citados, como podemos observar no quadro esquemático a seguir:



15.3. Avaliando Expressões:

A esta altura dos acontecimentos a curiosidade sobre como avaliar expressões usando conceitos definidos recursivamente já deve estar bastante aguçada. Não vamos, portanto retardar mais essa discussão. Apresentamos a seguir um modelo bastante simples para que possamos entender como avaliar expressões que usam conceitos definidos recursivamente. Novamente não precisaremos entender do funcionamento interno de um computador nem da maneira como uma determinada implementação de Python foi realizada. Basta-nos o conceito de redução que já apresentamos anteriormente.

Relembremos o conceito de redução. O avaliador deve realizar uma sequência de passos substituindo uma expressão por sua definição, até que se atinja as definições primitivas e os valores possam ser computados diretamente.

Vamos aplicar então este processo para realizar a avaliação da expressão ***fat 5***

passo	**Redução**	**Justificativa**
0	**fat 5**	expressão proposta
1	**5 * fat 4**	substituindo fat por sua definição geral
2	**5*(4 * fat 3)**	Idem
3	**5*(4* (3 * fat 2)**)	Idem
4	**5*(4*(3*(2 * fat 1))**)	Idem
5	**5*(4*(3*(2*(1 * fat 0)**	Idem
6	**5*(4*(3*(2*(1 * 1))))**	usando a definição específica
7	**5*(4*(3*(2*1)))**	usando a primitiva de multiplicação
8	**5*(4*(3*2)**	Idem
9	**5*(4*6)**	Idem
10	**5 * 24**	Idem
11	**120**	Idem

Surpreso(a)? Simples, não? É assim mesmo, bem simples. A cada passo vamos substituindo uma expressão por outra até que nada mais possa ser substituído. O resultado surgirá naturalmente. Mais tarde voltaremos ao assunto.

15.4. Recursão em Listas:

A esta altura deste curso já estamos certos que o uso de lista é indispensável para escrever programas interessantes. Em vista disso, nada mais óbvio que perguntar sobre o uso de recursão em listas. Veremos que o uso de definições recursivas em listas produz descrições simples, precisas e elegantes.

Já está na hora de alertar que os valores sobre os quais aplicamos os conceitos que queremos definir recursivamente possuem uma característica importantíssima, eles em si são recursivos.

Por exemplo, qualquer valor pertencente aos naturais pode ser descrito a partir da existência do zero e da função sucessor (suc). Vejamos como podemos obter o valor 5:

5 = suc(suc(suc(suc(suc 0)))))

As listas são valores recursivos. Podemos descrever uma lista da seguinte maneira:

Uma lista é:

1. a lista vazia;
2. um elemento seguido de uma lista

Esta natureza recursiva das listas nos oferece uma oportunidade para, com certa facilidade, escrever definições recursivas. A técnica consiste basicamente em:

1. Obter a definição geral: isto consiste em identificar uma operação binária simples que possa ser aplicada a dois valores. O primeiro deles é o **primeiro** da lista e o outro é um valor obtido pela aplicação do conceito em definição ao resto da lista;
2. Obter a definição independente, que se aplicará à base da recursão. Esta, em geral, é a ***lista vazia***;
3. Garantir que a aplicação sucessiva resto da lista levará à base da recursão.

Na Figura 15.2 ilustramos o processo recursivo de obter listas cada vez menores, detacando-se o primeiro elemento do restante da lista. Ao final do processo de aplicação sucessiva desta operação, obteremos a lista vazia ([ ]).

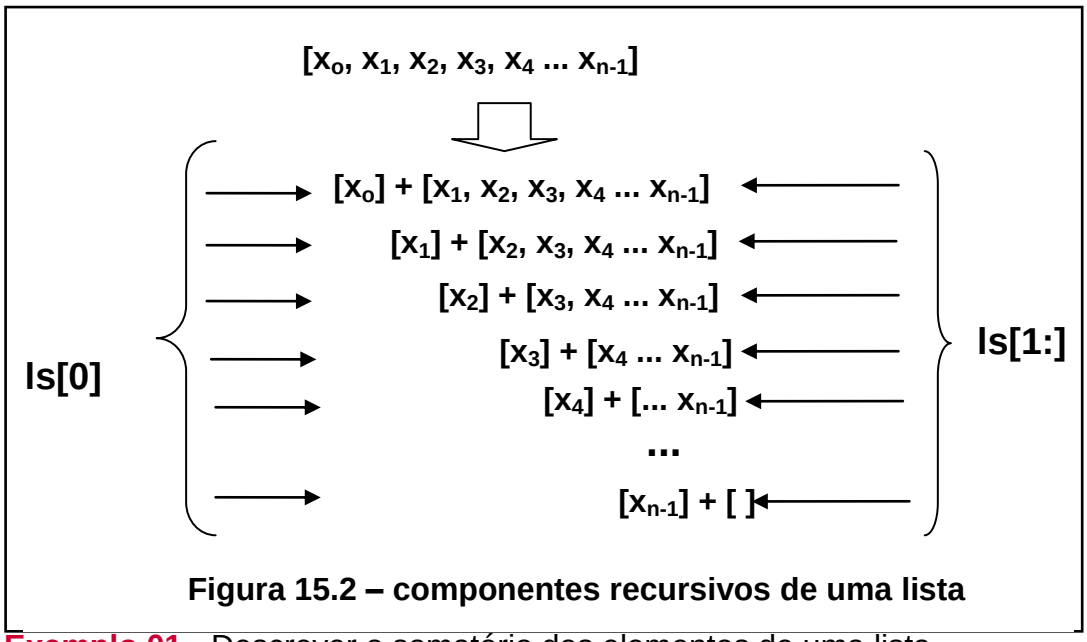


Figura 15.2 – componentes recursivos de uma lista

Exemplo 01 - Descrever o somatório dos elementos de uma lista.

Solução - Podemos pensar da seguinte maneira: o somatório dos elementos de uma lista é igual à soma do primeiro elemento da lista como o somatório do resto da lista. Além disso, o somatório dos elementos de uma lista vazia é igual a zero.

somat ([x_o, x_1, x_2, x_3, x_4 ... x_{n-1}])

x_o + somat([x_1, x_2, x_3, x_4 ... x_{n-1}])

x_0 + x_1 + somat([x_2, x_3, x_4 ... x_{n-1}])

x_0 + x_1 + x_2 + somat([x_3, x_4 ... x_{n-1}])

x_0 + x_1 + x_2 + x_3 + somat([x_4 ... x_{n-1}])

x_0 + x_1 + x_2 + x_3 + x_4 + somat([... x_{n-1}])

x_0 + x_1 + x_2 + x_3 + x_4 + ... + x_{n-1} + somat([ ])

x_0 + x_1 + x_2 + x_3 + x_4 + ... + x_{n-1} + 0

Figura 15.2 – desenvolvimento da computação de somatório dos elementos de uma lista

Vejamos então a codificação:

```
>>> def somat(lista):
      if lista == []: return 0
      else: return lista[0] + somat(lista[1:])

>>> somat(range(10))
45
```

Exemplo 02 - Descrever a função que determina o elemento de valor máximo uma lista de números.

Solução: O máximo de uma lista é o maior entre o primeiro elemento da lista e o máximo aplicado ao resto da lista. Uma lista que tem apenas um elemento tem como valor máximo o próprio elemento (Figura 15.3).

maximo([x_o, x_1, x_2, x_3, x_4 ... x_{n-1}])
maior(x_o , maximo([x_1, x_2, x_3, x_4 ... x_{n-1}]))
maior(x_o , maior(x_1, maximo([x_2, x_3, x_4 ... x_{n-1}])))
maior(x_o , maior(x_1, maior(x_2, maximo([x_3, x_4 ... x_{n-1}]))))
maior(x_o , maior(x_1, maior(x_2, maior(x_3, maximo([x_4 ... x_{n-1}])))))
maior(x_o , maior(x_1, maior(x_2, maior(x_3, maior(x_4, ... maximo([x_{n-2} , x_{n-1}])))))
maior(x_o , maior(x_1, maior(x_2, maior(x_3, maior(x_4, ... maior(x_{n-2} , maximo([x_{n-1}])))))
maior(x_o , maior(x_1, maior(x_2, maior(x_3, maior(x_4, ... maior(x_{n-2} , x_{n-1}))))))

Figura 15.3 – desenvolvimento da computação do elemento máximo de uma lista

A definição recursiva é apresentada a seguir:

```
>>> def maximo(lista):
      def maior(x, y):
            if x > y: return x
            else: return y
      if lista == []:  print "Funcao nao definida: lista vazia"
      elif len(lista) == 1: return lista[0]
      else: return maior(lista[0], maximo(lista[1:]))

>>> maximo([3,6,4,9,6,8])
9
```

Exemplo 03 - Descrever a função que verifica se um dado valor ocorre em uma lista também dada.

Solução: Podemos pensar da seguinte maneira: Um dado elemento k ocorre em uma lista se ele é igual ao primeiro elemento da lista ou se ele ocorre no resto da lista. Em uma lista vazia não ocorrem elementos quaisquer (Figura 15.4).

ocorre (k, [x_o, x_1, x_2, x_3, x_4 ... x_{n-1}])
k = x_o **or** ocorre (k, [x_1, x_2, x_3, x_4 ... x_{n-1}])
k = x_o **or** (k = x_1 **or** ocorre (k, [x_2, x_3, x_4 ... x_{n-1}]))
k = x_o **or** (k = x_1 **or** (k = x_2 **or** ocorre (k, [x_3, x_4 ... x_{n-1}])))
k = x_o **or** (k = x_1 **or** (k = x_2 **or** (k = x_3 **or** ocorre (k, [x_4 ... x_{n-1}]))))
k = x_o **or** (k = x_1 **or** (k = x_2 **or** (k = x_3 **or** (k = x_4) **or** ocorre (k, [... x_{n-1}])))))
...
k = x_o **or** (k = x_1 **or** (k = x_2 **or** (k = x_3 **or** (k = x_4) **or** ... ocorre k [x_{n-1}])))))
k = x_o **or** (k = x_1 **or** (k = x_2 **or** (k = x_3 **or** (k = x_4) **or** ... **or** (k = x_{n-1}) **or** ocorre (k, [ ])))))
k = x_o **or** (k = x_1 **or** (k = x_2 **or** (k = x_3 **or** (k = x_4) **or** ... **or** (k = x_{n-1}) **or** False)))))

Figura 15.4 – desenvolvimento da computação da ocorrência de um elemento em uma lista

Vejamos então a codificação:

```
>>> def ocorre(k, lista):
      if lista == []: return False
      else: return k == lista[0] or ocorre(k, lista[1:])


>>> ocorre(6, [3,6,4,9,6,8])
True
```

Exemplo 04 - Descrever a função que obtém de uma lista xs a sublista formada pelos elementos que são menores que um dado k :

Solução: Precisamos descrever uma nova lista, vamos denominá-la de ***menores***, em função de xs e de k. Quem será esta nova lista? Se o primeiro elemento de xs for menor que k, então ele participará da nova lista, que pode ser descrita como sendo formada pelo primeiro elemento de xs seguido dos menores que k no resto de xs. Se por outro lado o primeiro não é menor que k, podemos dizer que a lista resultante é obtida pela aplicação de menores ao resto da lista. Novamente a base da recursão é definida pela lista vazia, visto que em uma lista vazia não ocorrem elementos menores que qualquer k.

A codificação é apresentada a seguir:

```
>>> def menores(k, lista):
      if lista == []: return []
      elif k <= lista[0]: return menores(k, lista[1:])
      else: return [lista[0]] + menores(k, lista[1:])


>>> menores(6, [3,6,4,9,6,8])
[3, 4]
```

15.5. EXPLORANDO REUSO: Segundo o Professor George Polya, após concluir a solução de um problema, devemos levantar questionamentos a respeito das possibilidades de generalização da solução obtida. Dentro deste espírito, vamos explorar um pouco a solução obtida para o problema descrito a seguir.

Exemplo 5 (Sub-lista de números pares): Dada uma lista xs, desejamos descrever uma sublista de xs formada apenas pelos números pares existentes em xs.

Solução: Devemos considerar a existência de suas situações, como no problema de encontrar a sublista dos menores (Exemplo 4):

1. O primeiro elemento da lista é um número par, neste caso a sublista resultante é dada pela junção do primeiro com a sublista de pares existente no resto da lista.
2. O primeiro não é par. Neste caso a sublista de pares em xs é obtida pela seleção dos elementos pares do resto de xs.

Concluindo, tomemos como base da recursão a lista vazia, que obviamente não contém qualquer número.

Eis a solução em Python:

```
>>> def par(x): return x%2==0

>>> def slPares(lista):
      if lista == []: return []
      elif par(lista[0]): return [lista[0]] + slPares(lista[1:])
      else: return slPares(lista[1:])

>>> slPares([3,6,4,9,6,8])
[6, 4, 6, 8]
```

Vamos agora, seguindo as orientações do mestre Polya, buscar oportunidades de generalização para esta função. Podemos fazer algumas perguntas do tipo:

1. Como faria uma função para determinar a sublista dos números ímpares a partir de uma dada lista?
2. E que tal a sublista dos múltiplos de cinco?

Uma breve inspeção na solução acima nos levaria a entender que a única diferença entre as novas funções e a que já temos é a função que verifica se o primeiro elemento satisfaz uma propriedade, no caso presente a de ser um número par (even), conforme destacamos a seguir:

```
>>> def par(x): return x%2==0
>>> def slPares(lista):
      if lista == []: return []
      elif par(lista[0]): return [lista[0]] + slPares(lista[1:])
      else: return slPares(lista[1:])


>>> def impar(x): return x%2==1
>>> def slImpares(lista):
      if lista == []: return []
      elif impar(lista[0]): return [lista[0]] + slImpares(lista[1:])
      else: return slImpares(lista[1:])


>>> def mult5(x): return x%5==0
>>> def slMult5(lista):
      if lista == []: return []
      elif mult5(lista[0]): return [lista[0]] + slMult5(lista[1:])
      else: return slMult5(lista[1:])
```

Isto nos sugere que a função avaliadora pode ser um parâmetro. Pois bem, troquemos então o nome da função por um nome mais geral e adicionemos à sua interface mais uma parâmetro. Esta função, como sabemos, deverá ser do tipo Boolean. Vejamos

então o resultado da codificação, onde a propriedade a ser avaliada se converte em um parâmetro:

```
>>> def sublista(prop, lista):
      if lista == []: return []
      elif prop(lista[0]): return [lista[0]] + sublista(prop, lista[1:])
      else: return sublista(prop, lista[1:])

>>> sublista(par, [3,6,4,9,6,8])
[6, 4, 6, 8]

>>> sublista(impar, [3,6,4,9,6,8])
[3, 9]

>>> sublista(mult5, [3,6,4,9,6,8])
[]

>>> sublista(lambda x: x<5, [3,6,4,9,6,8])
[3, 4]

>>> sublista(lambda x: x>=5, [3,6,4,9,6,8])
[6, 9, 6, 8]
```

Observe que a função que havíamos anteriormente definido para determinar os elementos menores que um determinado valor k, da mesma forma que a função para determinar os maiores que k, está contemplada com a nossa generalização. As duas últimas avaliações no quadro acima ilustram a determinação da sublista dos valores menores que 5 e a dos maiores ou iguais a 5 usando a notação lambda.

Exercícios:

Descreva funções que utilizem recursão para resolver os problemas abaixo.

1. Obter a interseção de duas listas xs e ys.
2. inverter uma lista xs;
3. Eliminar os elementos de repetidos de uma lista.
4. Determinar a posição de um elemento x em uma lista xs, se ele ocorre na lista.
5. Dadas duas listas xs e ys, ordenadas em ordem crescente, obter a lista ordenada resultante da intercalação de xs e ys.