Algoritmos e Programação I: Aula 03*

Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
79070-900 Campo Grande, MS
`https://ava.ufms.br`

Sumário

1 Solucionando Problemas usando Computadores **1**
1.1 A Estrutura dos Computadores 2
1.2 Computadores e Funções 7
1.3 Funções e Algoritmos - Exemplos 10
1.4 Desenvolvendo Soluções Computacionais 16

2 Desenvolvimento de Sistemas **24**
2.1 Ciclo de Vida do Desenvolvimento de um Sistema 24

1 Solucionando Problemas usando Computadores

Na aula passada descrevemos o conceito de algoritmos e de como eles são implementados num computador. Na aula de hoje faremos uma associação mais detalhada da relação entre computadores, funções, algoritmos e programas. Os computadores são projetados para solucionar problemas dentro de um escopo bastante amplo. Com a incorporação do conceito de programa armazenado no projeto dos computadores, eles passaram a servir como máquinas de propósito geral, e com a utilização de programas, passaram a resolver vários tipos de problemas. De qualquer forma, para resolver um problema usando um computador, precisamos de um algoritmo que descreva "como" o problema será resolvido. Um algoritmo para resolver um problema usando um computador difere de outros tipos de algoritmos (uma receita para fazer arroz, aprender a andar de bicicleta, etc), pois a descrição dos passos do algoritmo deve ser de acordo com as funcionalidades de um computador. Um algoritmo computacional pode ser visto como uma função matemática, onde dada uma entrada (domínio da função) após um conjunto de operações (composição de funções intermediárias), obtemos uma saída (imagem da função), sendo que o conjunto de operações deve ser projetado de acordo com as funcionalidades do computador. Para que o algoritmo possa ser executado pelo computador, temos que implementá-lo numa linguagem de programação. Na descrição e implementação dos algoritmos deste curso, usaremos a linguagem de programação `Python` que possui instruções que implementam os passos de um algoritmo computacional.

*Este material é para o uso exclusivo da disciplina de Algoritmos e Programação I da FACOM/UFMS e utiliza as referências bibliográficas constantes na página da disciplina.

1.1 A Estrutura dos Computadores

Vamos agora descrever brevemente do ponto de vista operacional quais devem ser as funcionalidades que computador deve ter para resolver um problema algorítmico. Fundamentalmente, um computador deve ter hardware para implementar as seguintes funções:

- Ler uma entrada (input) de um dispositivo de entrada (teclado, mouse, cartão de memória) para a memória.
- Escrever a saída (output) da memória para um dispositivo de saída (monitor, cartão de memória, impressora).
- Executar operações (aritmética, comparações, etc.) em valores armazenados na memória.
- Controlar a sequência na qual as ações acima são executadas.

A utilidade do computador para resolver problemas é que ele pode executar essas funções simples de uma forma extremamente rápida e extremamente precisa.

No nível mais simples de hardware, os computadores são capazes de ler ou imprimir um único caractere. Com a utilização do software do sistema operacional e de linguagens de programação (*Python*, *C*, *Java*, etc) outras funcionalidades foram adicionadas para permitir a entrada e saída de quantidades agrupadas tais como números ou strings de caracteres. Desta forma os computadores tem a habilidade para entrada ou saída de um valor para uma variável, independentemente do tipo da variável.

Recursos externos ou dispositivos nos quais a entrada ou saída de dados ocorrem são conhecidos como fluxos.

Definição 1 *Um* ***fluxo*** (***stream***) *é uma sequência de valores que são normalmente acessados em ordem sequencial. Isto é, para acessar o n-ésimo valor em um fluxo, os* $n-1$ *valores precedentes devem ser acessados primeiro. Aqui o termo acesso aplica-se tanto para a leitura como para a escrita de um valor.*

A linguagem `Python` utiliza as funções `input()` e o `print()` para manipular os fluxos de entrada e saída. Outras funções (métodos) também são disponibilizadas para manipulação de fluxos. Posteriormente descreveremos os detalhes de como o `Python` manipula os fluxos de entrada e saída em um programa.

A memória do computador é um hardware feito com um grande número de circuitos eletrônicos. A memória é utilizada para armazenar valores individuais de dados enquanto um programa está sendo executado, como também as instruções que correspondem ao programa. Cada dado individual é associado com uma posição distinta da memória. Operacionalmente, podemos visualizar a memória como contendo um número de posições unicamente nomeadas, cada uma das quais é identificada com uma *variável* diferente no programa.

Uma noção básica em computação é a de *estado* do sistema, que pode se visto como uma fotografia com a descrição dos valores de todas as variáveis contidas na memória do computador em um dado instante durante a execução do programa. Por exemplo, no programa para calcular a soma de dois números inteiros, antes do Passo 1, os estado das variáveis `numero1`, `numero2` e `soma` não está definido. Após o Passo 1, (após o usuário fornecer os valores dos números, p. ex. `numero1 = 3` e `numero2 == 4`) o estado da memória mostra as variáveis relativas a `numero1` e `numero2` com os valores atribuídos (3 e 4, respectivamente) e a variável soma indefinida. Após o Passo 2, o estado das variáveis será: `numero1 == 3`, `numero2 == 4` e `soma == 7`. No algoritmo para calcular a área e o perímetro do quadrado, o estado da memória para as variáveis `lado`, `perimetro` e `area` antes do Passo 1, estão com

valores indefinidos. Após o Passo 1, (após o usuário fornecer o valor do lado, p. ex. `lado = 3.5`) o estado da memória mostra a variável relativa a lado com o valor atribuído (3.5) e as demais indefinidas.

Definimos a noção de estado do sistema mais formalmente como segue:

Definição 2 *Seja uma lista de variáveis $x_1, x_2, \ldots, x_n$ cujos tipos são definidos pelos conjuntos $S_1, S_2, \ldots, S_n$, respectivamente. Suponhamos também que os valores de x_1 é $s_1 \in S_1$, o valor de x_2 é $s_2 \in S_2$, e assim sucessivamente. Então a expressão*

$$x_1 == s_1 \ e \ x_2 == s_2 \ e \ \ldots \ e \ x_n == s_n$$

é denominado o estado das variáveis $x_1, x_2, \ldots, x_n$. O conjunto de todos os possíveis estados de $x_1, x_2, \ldots, x_n$ é conhecido como ***espaço de estado*** *dessas variáveis.*

Para um algoritmo implementado num computador por meio de um programa, os conteúdos da memória, entrada e saída definem o estado corrente do programa, constituindo a visão operacional de uma computação. Programas tipicamente terão espaço de estados enumerável, podendo ser extremamente grandes, mas geralmente finitos.

A capacidade de entrada e saída de informações para e da memória dá ao computador uma habilidade para interagir com seu ambiente. Mas é o poder de manipular a informação armazenada na memória que é que permite ao computador resolver uma grande variedade de problemas. As operações que um computador pode executar são bastante limitadas:

- Operações *Aritméticas*, usando os valores de variáveis naturais, inteiros e reais
- Operações *Relacionais* (lógicas), tais como comparação de valores de duas variáveis
- *Movimentar* um valor de uma posição da memória para outro, denominada atribuição
- Operações de *Controle*, as quais podem ser usadas para alterar a sequência em que os passos são executados, dependendo da natureza dos dados

O valores de variáveis naturais, inteiros e reais não possuem cardinalidade infinita em computadores (como o $\mathbb{N}$, $\mathbb{Z}$ e $\mathbb{R}$ tem em matemática), visto que a quantidade de espaço na memória disponível para cada valor individual é fixado em tamanho.

Lembre-se ainda que nem todas as operações podem ser efetuadas para todo valor. Isto é, cada valor é um membro de um tipo de dado (conjunto) o qual determina as operações válidas para aquele valor.

Note que computadores não possuem operações pré-definidas para fazer coisas do tipo "olhe para uma fotografia e reconheça a cena", "coloque uma lista de livros em ordem alfabética", etc. Por outro lado, usando algoritmos, os computadores podem ser programados para executar cada uma destas tarefas complexas por meio da subdivisão dessas tarefas em um (grande) número de passos bastante simples.

Abstração, como iremos ver, é o processo de dar a um grupo de ideias um único nome e então podemos simplesmente referenciar todo o grupo de ideias com a utilização desse nome. Podemos pensar uma abstração como sendo uma composição de funções.

Atribuição - Considere um programa com a variável `float x`, (*variável* `float` significa a mesma coisa que *variável do tipo* `float`). Suponhamos que o programa contenha uma atribuição da forma

```
x = 1.0
```

Esta instrução atribui o valor `1.0` para a variável `x`. (Note novamente que o uso de `=` não denota um teste matemático para igualdade na descrição de algoritmos. O par de símbolos `==` é reservado para o propósito de testar igualdade, como foi indicado na discussão anterior a respeito de estado.)

A atribuição também pode receber uma expressão aritmética, onde os termos dessa expressão podem conter variáveis que receberam valores (por leitura ou atribuição) anteriormente. Sejam as variáveis `a` e `b` do tipo inteiro (`int`).

```
a = 1
b = 2 * 3 - 5 + a
```

Nesse caso, o valor inteiro 1 é atribuído à variável `a` e num passo seguinte, o valor inteiro do lado direito do sinal de `=`, `2 * 3 - 5 + a`, é computado e atribuído à variável `b`. Nesse caso, primeiramente o valor da variável `a` é buscado na memória e substituído na expressão (`2 * 3 - 5 + 1`). Para computar o valor da expressão são usadas as regras de precedência utilizadas na matemática. Primeiro multiplicações, divisões e resto de divisão inteira e finalmente adições e subtrações (`6 - 5 + 1`). Quando mais de um operador tiver a mesma precedência, sempre computar na ordem da esquerda para a direita (`1 + 1` e finalmente `2`). Podemos usar parenteses para mudar a ordem em que as operações são efetuadas. No caso de haver parenteses na expressão, resolvemos primeiramente as expressões que estiverem entre parenteses, solucionando inicialmente os parenteses mais internos.

Exemplo - Quando se tratar de divisão de números inteiros, é importante observar que o resultado da divisão é um número inteiro. Sejam as variáveis `a` e `b` do tipo inteiro.

```
a = 3
b = 10//a
```

Na primeira linha é atribuído o valor inteiro `3` para a variável `a`. Na segunda linha, o lado direito do sinal de `=`, `10//a`, será avaliado e o resultado será atribuído à variável `b`. Inicialmente, o valor da variável `a` é buscado na memória e substituído na expressão, `10/3`, após é efetuada a divisão inteira `10/3`, cujo resultado é `3`, e esse valor será armazenado (atribuído) na posição da memória relativa a variável `b`.

Exemplo - Calcule o valor da variável real (tipo `float`) `valor` na atribuição:
`valor = 3.0 + 4.0 * 5.0`

```
valor = 3.0 + 4.0 * 5.0  (0)

valor = 3.0 + 20.0 (1) * tem prioridade com relação a +
valor = 23.0 (2)
```

Exemplo - Calcule o valor da variável `valor` na atribuição:
`valor = 8.0 / 4.0 * 5.0`

```
valor = 8.0 / 4.0 * 5.0  (0)

valor = 2.0 * 5.0 (1) * e / mesma prioridade, usar esquerda para direita
valor = 10.0 (2)
```

A capacidade de ler e imprimir informações na memória e a manipular os valores das variáveis na memória não é suficiente para executar abstrações computacionais significativas. Controle do sequenciamento e seleção na qual as atribuições são feitas é também importante.

Considere a seguinte sequência de instruções:

```
i = 10
k = 100
```

Não faz diferença em qual ordem executaremos estas duas atribuições, visto que duas variáveis diferentes são afetadas. Independentemente da ordem em que elas forem executadas, o resultante estado é `i == 10` e `k == 100` é o mesmo.

Agora considere o seguinte trecho de programa:

```
i = 0
i = 5
j = 10//i
```

Aqui, a ordem da execução das duas últimas instruções tem um grande impacto no estado resultante. Executados na ordem indicada, temos o estado `i == 5` e `j == 2`. Mas se invertermos a ordem de execução, então o estado é indefinido (visto que a atribuição `j = 10//0` não pode ser computada).

Considere também o problema de atribuir o maior valor de duas variáveis para uma terceira variável usando a instrução `if`, `then (:)` e `else`:

```
if var1 > var2:
   var3 = var1
else
   var3 = var2
```

Isto é, desejamos executar uma atribuição diferente dependendo do presente estado da memória (os valores atuais de `var1` e `var2`). Desta forma o estado final irá depender do estado inicial (um possível estado final é `var3 == var1` enquanto o outro é `var3 == var2`).

Estes exemplos indicam que necessitamos estar aptos para controlar o fluxo de nossos programas; isto é, programas devem ter uma forma de sequenciar as várias instruções que manipulam os valores de variáveis na memória. A última capacidade operacional do computador é esta habilidade.

Existem três instruções fundamentais de controle de instruções:

- *Sequencial* - Esta é o *default*; as instruções são normalmente executadas sequencialmente, na ordem em que elas aparecem na listagem programa.
- *Seleção* - Esta forma de instrução de controle permite-nos fazer uma escolha entre uma ou mais possibilidades. Por exemplo, podemos determinar o valor absoluto da variável `var1` usando a instrução de seleção `if`, `:` e `else`:

  ```
  if var1 < 0.0:
     absoluto = -var1
  else
     absoluto = var1
  ```

- *Iteração* (ou repetição) - Esta forma de instrução de controle permite-nos repetir uma série de instruções 0 ou mais vezes, dependendo de uma condição a ser verificada.

```
contador = 10
while contador >= 0:
   print(contador)
   contador = contador - 1
# fim
```

Uma grande parte do processo de programação é o de entender perfeitamente as diferenças entre estas três formas de controle.

Vamos agora analisar o o ocorre no computador na medida que executado os passos de um dado algoritmo. Utilizando o algoritmo `Soma` descrito na aula passada, descreveremos como as funcionalidades de um computador são utilizadas para executar os passos de um algoritmo:

```
# Passo 1.  Leia número1 e número2
numero1 = int(input())
numero2 = int(input())
```

- Dispositivo de entrada para a memória do computador.

```
# Passo 2.  Compute soma dos números
soma = numero1 + numero2
```

- Operações na memória do computador.

```
# Passo 3.  Imprima a soma
print(soma)
```

- Memória para o dispositivo de saída do computador.

Também podemos descrever o estado do nosso sistema a cada passo na execução do algoritmo:

- Os valores das variáveis `numero1`, `numero2` e `soma` estão indefinidos.

```
# Passo 1.  Leia número1 e número2
numero1 = int(input())
numero2 = int(input())
```

- `numero1` e `numero2` agora têm valores; a variável `soma` não tem valor atribuído.

```
# Passo 2.  Compute soma dos números
soma = numero1 + numero2
```

- `número1`, `número2` e `soma` agora tem valores atribuídos.

```
# Passo 3.  Imprima a soma
print(soma)
```

Agora utilizaremos o exemplo do cálculo do Perímetro e Área para descrever como as funcionalidades podem ser utilizadas para implementar os passos de um algoritmo:

```
# Passo 1.  Leia o lado do quadrado
lado = float(input())
```

- Dispositivo de entrada para a memória do computador.

```
# Passo 2.  Calcule o perímetro do quadrado
perimetro = 4 * lado
```

- Operações na memória do computador.

```
# Passo 3.  Calcule a área do quadrado
area = lado * lado
```

- Operações na memória do computador.

```
# Passo 4.  Imprima o perímetro e a área
print(perimetro, area)
```

- Memória para o dispositivo de saída do computador.

Também podemos descrever o estado do nosso sistema a cada passo na execução do algoritmo:

- Os valores das variáveis `lado`, `area` e `perimetro` estão indefinidos.

```
# Passo 1.  Leia o lado do quadrado
lado = float(input())
```

- `lado` agora tem um valor; as variáveis `area` e `perimetro` não tem valores atribuídos.

```
# Passo 2.  Calcule o perímetro do quadrado
perimetro = 4 * lado
```

- `lado` e `perimetro` agora tem valores; `area` não tem valor atribuído.

```
# Passo 3.  Calcule a área do quadrado
area = lado * lado
```

- `lado`, `perimetro` e `area` agora tem valores atribuídos.

```
# Passo 4.  Imprima o perímetro e a área
print(perimetro, area)
```

1.2 Computadores e Funções

Analisamos a visão operacional do computador em termos das propriedades gerais que os computadores possuem. Essa visão que discutimos é uma visão distinta de uma mais orientada para hardware que será vista na disciplina de **Introdução a Sistemas de Computação**, onde são estudados os sinais eletrônicos e como eles são usados para projetar computadores. A forma funcional como estamos apresentando os computadores é muito próxima da noção matemática de função a qual, por sua vez, é fundamental para toda computação. Vamos agora associar a noção de função com algoritmos e programas.

Vamos rever alguns conceitos básicos sobre funções que serão vistos na disciplina de **Fundamentos de Teoria da Computação**.

Definição 3 *Sejam X e Y conjuntos. Uma* **função** *de um conjunto X para um conjunto Y é definida como um conjunto G de pares ordenados (x, y) tal que $x \in X$ e $y \in Y$, e todo elemento de X é o primeiro componente de exatamente um par ordenado de G. Isto é, para todo $x \in X$, existe exatamente um elemento $y \in Y$ tal que o par ordenado (x, y) pertence ao conjunto de pares definindo a função f. [Wikipedia]*

Usaremos a notação $f : X \to Y$ para denotar a função f. Os conjuntos X e Y são denominados o domínio e o contra-domínio da função f, e o conjunto G é chamado o **gráfico** da função f. Para (x, y) em G, dizemos que y é a imagem de x com respeito a função f, ou o valor de f aplicado ao *argumento* x e denotamos por $y = f(x)$.

Tanto o domínio de uma função como a imagem podem ser multidimensionais.

Definição 4 *O* **produto cartesiano** $X_1 \times X_2 \ldots \times X_n$ *de $n > 1$ conjuntos $X_1 \ldots X_n$ é o conjunto de todas n-uplas $(x_1, \ldots, x_n)$ tal que $x_i \in X_i$ para todo $1 \leq i \leq n$*

Portando, podemos definir uma função n-dimensional

$$f : U \to V$$

onde $U \subseteq X_1 \times \ldots \times X_n$ e $V \subseteq Y_1 \times \ldots \times Y_m$.

Exemplo - Considere o problema de dados dois números reais x e y, calcular a sua soma $x + y$. Podemos modelar esse problema com uma função $f(x, y) = x + y$, $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$.

Exemplo - Considere o problema de dados dois números inteiros distintos u e v, ordenar os números u e v em ordem crescente. Nesse caso, esse problema pode ser modelado com uma função $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}$.

Várias funções matemáticas estão disponíveis numa linguagem de programação. No *Python* muitas delas estão na biblioteca/módulo `math`. Além disso, podemos utilizar várias outras funções como `ord` e `chr` para calcular o código ASCII (ou Unicode) de um dado caractere e o caractere correspondente a um código ASCII (ou Unicode), respectivamente.

Exemplo - Funções um-a-um e inversas (*serão vistas em Fundamentos de Teoria da Computação*) são muitas vezes usadas em codificação. O *American Standard Code for Information Interchange* (ASCII) define uma equivalência numérica entre 0 e 127 para cada caractere regular ou caractere de controle que pode ser digitado em um teclado padrão de computador. Esta função é um-a-um, e é implementada em *Python* usando `ord`. Isto é, para qualquer caractere `x` que apareça no teclado, `ord(x)` corresponderá a um inteiro variando de 0 a 127 que corresponda a ele no conjunto ASCII. Alguns valores numéricos de caracteres do teclado são dados abaixo:

Python
`ord('0') == 48`
`ord('A') == 65`
`ord('=') == 61`
`ord('a') == 97`

A lista completa dos caracteres ASCII e seus valores inteiros correspondentes pode ser visto em Código ASCII.

Portanto as funções `ord` e `(chr)` implementam o esquema de codificação uniforme ASCII para caracteres do teclado. Este esquema é usado pela maior parte dos fabricantes de hardware e software para computadores. A existência de tais facilidades de padronização facilita o intercâmbio entre diferentes tipos de computadores. No Python, as funções `ord` e `chr` usam a codificação do Unicode. Os valores do Unicode no intervalo $[0, 127]$ são os mesmos do código ASCII.

Note que, para funcionar propriamente, `ord` tem que ser uma função um-a-um, pois se um caractere for atribuído para dois códigos numéricos distintos, não haveria uma forma consistente de decifrar o código. Visto que `ord` é um-a-um, ela têm uma função inversa correspondente na linguagem *Python*. chamada `chr` (caractere). Se `n` é um inteiro variando de 0 e 127, então a expressão `chr(n)` retorna o caractere correspondente no teclado para esse inteiro. Por exemplo:

Python
`chr(48) =='0'`
`chr(65) =='A'`
`chr(61) =='='`
`chr(97) =='a'`

Note que, `ord` e `chr` são pares de funções inversas uma da outra, sejamas expressões abaixo são sempre verdadeiras em *Python*:

```
chr(ord(varx)) == varx
```

e

```
ord(chr(n)) == n
```

para qualquer inteiro `n` variando de 0 a 127 e qualquer caractere `varx`.

A capacidade do computador em manipular caracteres possibilita uma aplicação muito importante em computação, a codificação e decodificação de mensagens secretas Essa área é denominada *Criptografia*.

Exemplo - Um dos métodos mais antigos de criptografia é o *Cifrador de César*. O Cifrador de César é um esquema de codificação na qual cada letra da mensagem é substituída por uma letra diferente do alfabeto, digamos a terceira letra seguindo-a. Desta forma a mensagem

```
GISELEBUDCHENDESFILOUNAVILAISABEL
```

é codificada no cifrador de César como

```
JLVHOHEXGFKHQGHVILORXQDZLODLVDEHO
```

Portanto o cifrador de César é uma permutação das letras do alfabeto e desta forma é uma função um-a-um cujo domínio e imagem é o alfabeto romano maiúsculo. Decodificando uma tal mensagem é equivalente a descoberta da função inversa que foi usada para codifica-la inicialmente.

A desvantagem do cifrador de César é que ele é relativamente fácil de decodificar. Podemos pensar em um código que é mais difícil de decodificar por meio da utilização de uma permutação aleatória e substituindo os caracteres usando essa permutação. Usando a tabela de substituição

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
JSATERNICFHPLUGWVYDZOMXQBK
```

permite-nos codificar a mensagem

```
GISELEBUDCHENDESFILOUNAVILAISABEL
```

como

```
NCDEPESOTAIEUTEDRCPGOUJMCPJCDJSEP
```

Se a mensagem for longa o suficiente, mesmo um cifrador aleatório pode ser decodificada pela contagem da frequência das letras e emparelhando-as com as frequências das ocorrências das letras em no idioma em que a frase foi escrita. Alguns sistemas mais sofisticadas de codificação aplicam funções de codificação para blocos de letras.

Podemos também pensar num algoritmo (programa) como sendo uma função f, pois **todas as possíveis instâncias da entrada** formam o domínio e a **saída** é a imagem da função. A descrição da função f é dada por $f = f_1 \circ f_2 \circ \cdots \circ f_n$, onde f_i pode ser associado ao `Passo i` do algoritmo.

1.3 Funções e Algoritmos - Exemplos

Agora vamos descrever alguns exemplos de funções e algoritmos que envolvem o processamento de números, texto e gráficos. Os exemplos são bem simples, mas eles formam os fundamentos das ideias que serão utilizadas em problemas mais complexos.

Exemplo 1. Nosso primeiro exemplo é bem simples. Consiste apenas de uma função que dado um valor real computa o quadrado desse número. O Domínio da função é dado pelos reais e a imagem da função é dada pelos reais maiores ou iguais a zero. A função é dadas por:

```
quadrado(número) = número*número
```

A seguir descreveremos o algoritmo que implementa a função que computa o quadrado de um número:

```
1  # Algoritmo: Quadrado
2  # Este algoritmo lê um número real e computa e imprime o seu
3  # quadrado.
4  # descrição das variáveis utilizadas
5  # float numero, quadrado
6
7  # pré: número
8
9  # Passo 1. Leia um número
10 numero =  float(input()) # lê o fluxo e converte para float
11 # Passo 2. Calcule o quadrado do número
12 quadrado = número * número
13 # Passo 3. Imprima o quadrado do número
14 print(quadrado)
15 # pós: quadrado == número * número
16 # fim do Algoritmo
```

Posteriormente iremos explicar com mais detalhes os significados de `pré` e `pós`. Aqui eles podem ser interpretados como sendo o domínio e a imagem, respectivamente, da função $y = Quadrado(x)$.

Exemplo 2. Alguns algoritmos podem envolver mais de uma função. O exemplo do cálculo do perímetro e da área de uma circunferência envolve duas funções. O Domínio das duas funções é dado pelos reais maiores que zero e a imagem das funções também é dada pelo pelos reais positivos. As funções são dadas por:

```
Perímetro(raio) = 2 * pi * raio
Área(raio) = pi * raio * raio
```

onde `pi = 3.1415...`.

Uma outra forma de ver esse algoritmo é usando uma função $f : \mathbb{R} \to \mathbb{R}^2$, tal que $f(raio) = (2 * \pi * raio, \pi * raio * raio)$.

O algoritmo que implementa essas soluções pode ser dado por:

```
1  # Algoritmo: Perímetro_ÁreaC
2  # Este algoritmo lê o valor do raio de uma circunferência e
3  # calcula e imprime o perímetro e a área da circunferência.
4  # descrição das variáveis utilizadas
5  # float raio, perimetro, area
6
7  # pré: raio
8
9  # Passo 1. Leia o raio da circunferência
10 raio =  float(input()) # lê o fluxo e converte para float
11 # Passo 2. Calcule o perímetro e a área da circunferência
12 # Passo 2.1. Calcule o perímetro da circunferência
13 perimetro = 2 * pi * raio
14 # Passo 2.2. Calcule a área da circunferência
15 area = pi * raio * raio
16 # Passo 3. Imprima o perímetro e a área da circunferência
17 print(perimetro, area)
18
19 # pós: perímetro == 2* pi * raio && area == pi * raio * raio
20 # fim Algoritmo Perímetro_Área
```

Exemplo 3. Considere agora a função $f : \mathbb{N}^3 \to \mathbb{N}$, que dado os valores de uma medida de tempo em horas, minutos e segundos, computa o valor totak em segundos. Nesse caso, o domínio da função é dado pelo conjunto das ternas ordenadas em $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$, `(horas, minutos, segundos)` e a imagem é dada por $\mathbb{N}$. A função que calcula o total de segundos é definida como:

```
Tempo_Segundos(horas, minutos, segundos) =
                    60 * 60 * horas + 60 * minutos + segundos
```

Em geral quando um programa é utilizado para implementar uma função, seu conjunto de entradas admissíveis (neste caso todas as ternas `(horas, minutos, segundos)` correspondem ao domínio da função; seu conjunto de saídas ($\mathbb{N}$ neste caso) corresponde a noção

de Imagem; e seus passos definem as regras procedimentais de correspondência entre cada elemento do domínio e um elemento individual da imagem.

Dado uma medida de tempo em horas, minutos e segundos, como seriam os passos necessários para computar o total em segundos dessa medida de tempo. A entrada é a quantidade de horas, minutos e segundos e a saída e total em segundos dado pela quantidade de horas multiplicada por 60×60 somado com a quantidade de minutos multiplicado por 60 e somado com a quantidade de segundos.

A seguir descreveremos os detalhes do algoritmo que implementa essa função:

```
1  # Algoritmo: Tempo_Segundos
2  # Este algoritmo lê uma medida de tempo, dada em horas,
3  # minutos e segundos e calcula seu equivalente em segundos.
4  # descrição das variáveis utilizadas
5  # int horas, minutos, segundos, totalSegundos
6
7  # pré: horas, minutos, segundos > 0
8
9  # Passo 1. Leia a entrada horas, minutos, segundos
10 horas =  int(input()) # lê o fluxo e converte para int
11 minutos =  int(input()) # lê o fluxo e converte para int
12 segundos =  int(input()) # lê o fluxo e converte para int
13 # Passo 2. Calcule o total de segundos;
14 totalSegundos =  60 * 60 * horas +  60 * minutos + segundos
15 # Passo 3. Imprima o total de segundos
16 print(totalSegundos)
17
18 # pós: totalSegundos == 3600*horas + 60 * minutos + segundos
19 # fim Algoritmo Tempo_Segundos
```

A **entrada** pode ser uma medida de tempo qualquer, dada em horas, minutos e segundos, tal como

```
3
34
45
```

Esses valores podem ser armazenados em três variáveis do tipo `int`.

Se os passos 1, 2 e 3 (**processo**) forem executados cuidadosamente a saída para a entrada dada acima será

```
12885
```

Juntos, essa entrada e essa saída representam uma **instância** particular do problema do tempo em segundos.

Exemplo 4. Vamos agora ilustrar o caso em que o domínio da função é um conjunto de caracteres. Temos que definir com mais detalhes o que entendemos por conjuntos específicos de caracteres. Considere uma função que dada uma palavra, computa o tamanho da palavra. Como usaremos um computador para implementar a função, temos que especificar melhor os conceitos de caracteres e palavras. Os computadores utilizam alfabetos específicos, com caracteres que podem ser visíveis ou não. Para simplificar, consideraremos uma **palavra**

como sendo um conjunto de caracteres $c_0c_2 \ldots c_{k-1}$, onde cada c_i é uma letra do alfabeto da língua portuguesa. Toda palavra é precedida de um caractere espaço e após a palavra também tem um caractere espaço. Necessitamos também definir o que significa **tamanho de uma palavra**. Definimos **tamanho de uma palavra** o número de caracteres da palavra. De posse das definições de palavra e tamanho de uma palavra, podemos definir a função $f : \mathcal{P} \to \mathbb{N}$ que associa um número inteiro positivo (número de caracteres) a cada palavra do conjunto $\mathcal{P}$. O domínio da função é um dado conjunto de palavras $\mathcal{P}$ e a imagem é um conjunto de números inteiros. Para simplificar, podemos usar a função do Python que computa o tamanho de uma palavra:

```
len(palavra) == número de caracteres da palavra
```

O algoritmo abaixa implementa a função:

```
1  #Algoritmo: Tamanho_Palavra
2  # Este algoritmo lê uma palavra, computa o número de caracteres
3  # da palavra e imprime o resultado
4  # descrição das variáveis utilizadas
5  # string palavra
6  # int tamanho
7
8  # pré: c0c1...ck
9
10 # Passo 1. Leia os caracteres de uma palavra
11 palavra = input()
12 # Passo 2. Compute o tamanho da palavra
13 tamanho = len(palavra) # a função len() computa o número de caracteres
14 # Passo 3. Imprima o tamanho da palavra
15 print(tamanho)
16
17 # pós: palavra == c0c1...ck && tamanho(palavra) == k+1
18 # fim Algoritmo Tamanho_Palavra
```

A **entrada** é uma palavra qualquer (de acordo com a definição acima), dada por um conjunto de caracteres, tal como

```
internet
```

Se os passos 1, 2 e 3 (**processo**) forem executados corretamente a **saída** do problema para entrada acima será o número de caracteres da palavra `internet`.

```
8
```

Juntos, essa entrada e essa saída representam uma **instância** particular do problema de computar o tamanho de uma palavra.

Exemplo 5. O mesmo pode ser feito para componentes gráficos. Abordaremos o problema de imprimir um pixel numa dada cor na tela do computador. Da mesma forma que no problema anterior, temos que especificar em mais detalhes o que é um pixel e o que significa "na tela do computador". A impressão de um pixel na tela do computador depende

de como manipulamos o ambiente gráfico de um dado computador. Por outro lado, essa manipulação pode depender do sistema operacional e da linguagem utilizada. O mesmo problema ocorre com a cor. Posteriormente veremos em detalhes como tratar esses dois problemas. Para simplificar, consideraremos que a obtenção de uma janela gráfica, impressão de um pixel, e cores sejam primitivas dadas. Um **pixel** (aglutinação de *Picture* e *Element*, ou seja, elemento de imagem, sendo *Pix* a abreviatura em inglês para *Picture*) é o menor elemento num dispositivo de exibição (como por exemplo um monitor), ao qual é possível atribuir-se uma cor. De uma forma mais simples, um pixel é o menor ponto que forma uma imagem digital, sendo que o conjunto de milhares de pixeis formam a imagem inteira. Os pixeis possuem um endereço que é dado pelas suas coordenadas físicas no dispositivo de exibição. As **cores** num dispositivo de exibição podem ter vários padrões (p.ex. RGB, CMYK, etc). Dadas as definições acima, consideraremos nossa **entrada** como sendo um par de coordenadas (x, y) do pixel no dispositivo e o nome de uma cor. A **saída** do nosso problema será um pixel da cor da entrada mostrado na coordenada (x, y) do dispositivo de exibição. Nesse caso, nossa função recebe uma coordenada (x, y) em $\mathbb{Q} \times \mathbb{Q}$ e uma cor e retorna um pixel com a cor dada na tela do computador. A função pode ser definida como $f : \mathbb{Q} \times \mathbb{Q} \times \mathbb{I} \rightarrow \mathbb{P}$ onde $\mathbb{I}$ é o conjunto das cores numa dada codificação e $\mathbb{P}$ é o conjunto dos pixeis na tela de um computador.

```
f(x,y,a) ==  pixel de cor a de coordenadas (x,y) na tela do computador
```

No caso da impressão de um pixel na tela do computador, temos alguns detalhes que precisamos definir antes de imprimir um pixel na tela do computador usando Python. Inicialmente, precisamos definir uma janela gráfica para que o pixel possa ser mostrado. Para isso, podemos usar uma das bibliotecas disponíveis. Neste exemplo usaremos a biblioteca `pillow` (`PIL`). A biblioteca PIL possui um módulo que permite a geração e apresentação de imagens. Neste primeiro exemplo iremos omitir os detalhes e nos concentraremos nos métodos (funções) que podem ser usados nesse módulo para a geração e exibição de um pixel.

O algoritmo abaixo descreve a utilização da função (método) `putpixel()`:

```
1  # Algoritmo: Pixel
2  # Este programa utiliza o módulo pillow (PIL).
3  # Ele gera e imprime um pixel na tela.
4  # Declaração dos módulos utilizadas
5  from PIL import Image
6  # início do módulo principal
7  # descrição das variáveis utilizadas
8  # int x, y
9
10 # pré: x y
11
12 # Passo 1. Defina o tamanho da imagem e leia o ponto
13 # Passo 1.1. Defina o tamanho da imagem
14 largura = 1200
15 altura = 800
16 # Passo 1.2. Defina o padrão de cores
17 padrao = 'RGB'
18 # Passo 1.3. Cria uma nova imagem com fundo branco.
19 imagem = Image.new(padrao, (largura,altura), color='white')
```

```
# Passo 1.4 Leia um ponto (x,y)
x = int(input())
y = int(input())
# Passo 2. Gere o pixel vermelho de coordenadas (x,y) na imagem
imagem.putpixel((x,y),(255,0,0))
# Passo 3. Mostre a imagem resultante
imagem.show()

# pós: um pixel vermelho na tela
fim Algoritmo
```

A **entrada** é uma coordenada inteira (x, y), onde $0 \leq x \leq 1200$ e $0 \leq y \leq 800$ (de acordo com as especificações do imagem):

```
600
400
```

Se os passos 1, 2 e 3 (**processo**) forem executados corretamente a saída para a entrada dada acima será a imagem abaixo:

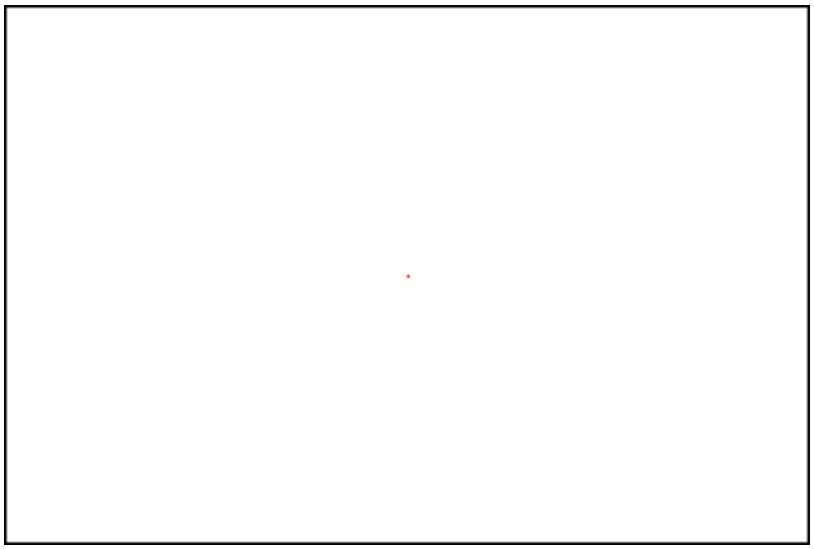

Figura 1: Pixel vermelho na posição (600,400)

Juntos, essa entrada e essa saída representam uma **instância** particular do problema da impressão de um pixel. A exibição de um único pixel não dá uma visualização muito boa. Existem várias outras bibliotecas para a manipulação de pixels. Inicialmente usaremos a biblioteca PIL (pillow). Também é possível utilizar a biblioteca `Turtle` para manipulação de objetos gráficos. Nas próximas aulas descreveremos outros exemplos de soluções computacionais usando Python e as bibliotecas PIL e Turtle.

O principal objetivo desta disciplina é o de desenvolver uma metodologia na qual problemas possam ser solucionados por meio de um desenvolvimento sistemático de programas de computador, tais como os descritos acima para o quadrado de um número, o perímetro e a área de uma circunferência, calculo do total de segundos, determinar o tamanho de uma palavra e a impressão de um pixel. O desenvolvimento de tais programas requerem do aluno a aprendizagem de várias habilidades simultaneamente. O aluno deve aprender a desenvolver algoritmos e programas usando uma metodologia que consiga traduzir as instruções de problemas algoritmos em especificações precisas e finalmente, programas completos. Além disso, o aluno deve aprender os detalhes da linguagem de programação *Python*.

1.4 Desenvolvendo Soluções Computacionais

O **Desenvolvimento de Soluções Computacionais** é um processo com vários passos que envolvem o entendimento correto do problema, o desenvolvimento da solução, e a codificação e teste do programa. Essa atividade é denominada *solução algorítmica de problemas*. Uma característica fundamental da solução algorítmica de problemas é que o nível de precisão - tanto na descrição do problema a ser solucionado quanto na codificação da solução proposta - deve ser bastante detalhado e rigoroso. Por exemplo, a descrição dos três passos do algoritmo do cálculo do perímetro e área que descrevemos na seção anterior não é suficientemente precisa para ser implementada em um computador. Todo o detalhamento descrito no programa completo que apresentamos a seguir é necessário por duas razões:

- O problema, suas limitações, e sua solução deve ser claramente descrito para um leitor humano.
- A solução deve resolver completamente o problema - isto é, deve equipar o computador com os meios para "entregar" uma resposta correta para *todas* as possíveis variações na entrada.

No problema do `PerímetroÁrea`, a imprecisão na descrição do processo deixa sem resposta um grande número de questões: Devemos prever lado igual a zero? E se o usuário informar um lado com valor negativo? Se lado igual a zero ou negativo não for uma entrada válida, devemos esclarecer a descrição do processo tal que essa restrição seja explícita. Se lado igual a zero ou negativo forem entradas válidas, então devemos modificar a descrição do processo tal que uma mensagem de saída apropriada (ao invés do perímetro e área) possa ser impressa. Um dos conceitos mais difíceis para o cientista da computação iniciante compreender é que a execução de um programa por um computador é um processo inteiramente mecânico. Isto é, o computador não "entende" o programa que ele está executando ao passo que os humanos conseguem entender o que está se passando. Humanos não seguem tais regras estritas, mas muitas vezes intuem o que se pretende quando eles decifram uma instrução. Esta diferença é causada por dois fatores. Primeiro, a mente humana é muito mais sofisticada que qualquer computador. Segundo, o entendimento humano funciona com base do contexto compartilhado entre quem fala e quem está ouvindo. Não existe tal contexto compartilhado entre o programador e o computador. Esta breve discussão enfatiza a necessidade de uma maneira mais formal, mais estilizada de descrever problemas algoritmos e suas soluções, uma que não deixe espaço para o computador "intuir" as intenções do programador quando descreve uma computação. Em outras palavras, necessitamos usar "linguagens de programação" quando descrevemos algoritmos para uma solução computacional. Além disso, problemas algoritmos tipicamente possuem várias características complexas, algumas das quais são inovativas por natureza. Isto sugere que o processo de solução algorítmica de problemas envolve um método sistemático ao invés de simplesmente sentar-se e escrever código *Python* diretamente; um pensamento inicial é necessário para descrever o problema, projetar um algoritmo, e desenvolver o programa resultante em uma forma passo-a-passo cuidadosa . Frequentemente, na prática, uma solução para um problema algorítmico grande é executado por uma equipe de pessoas trabalhando juntas ao invés de ser executado por uma pessoa trabalhando sozinha.

Boas soluções para problemas compartilham três características que são verdadeiramente fundamentais: legibilidade, correção, e eficiência.

- *Legibilidade* significa que qualquer um que esteja familiarizado com *Python* e com a aplicação para o qual o problema está sendo elaborado pode rapidamente entender o algoritmo com uma leitura cuidadosa do texto do programa acompanhado de suas especificações e comentários, mesmo se o problema é complexos e o programa é grande.

- *Correção* significa que todas as execuções do programa darão resultados válidos para todas as entradas que as especificações do programa admitem.

- *Eficiência* significa que o projeto do programa não desperdiça recursos computacionais - medidos pela quantidade de tempo e espaço na memória que ele usa - enquanto computando sua saída. Eficiência é desejável para minimizar o custo total e maximizar a utilização total das aplicações no computador.

Estas três características dos programas de computador tendem a dar aos programas uma identidade dual. Por um lado, a legibilidade dos programas permite que eles sirvam como *explicação* das soluções dos problemas algorítmicos. Por outro lado, a confiabilidade e eficiência destes mesmos programas permitem que eles sirvam como *mecanismos* por meio dos quais o computador pode ser usado para resolver problemas. Esta identidade dual é aparente no programa `perimetroareaQ.py`, onde a documentação dada pelos comentários (as linhas que são precedidas de `#`) permite a legibilidade e o resto do programa define os mecanismos computacionais para computar o `PerimetroArea`.

Em geral, o desenvolvimento de bons programas requer uma metodologia abrangente e unificada que possa, com confiabilidade, ser usada para resolver problemas em uma grande variedade de áreas de aplicação. Nesta disciplina desenvolvemos uma metodologia para solução algorítmica de problemas baseada na *MAPS* (*m*ethodology for *a*lgorithmic *p*roblem *s*olving), [Tucher et al.]. Vamos nos concentrar mais nos aspectos referentes a compreensão, projeto e implementação da solução para um problema algoritmo. Como usaremos a linguagem Python e em várias situações as bibliotecas disponíveis, daremos mais ênfase no entendimento do problema, na descrição da saída e das estruturas necessárias de entrada para que a saída esperada seja alcançada. Usaremos refinamentos sucessivos para descrever os passos do algoritmo até que eles possam ser implementados por um conjunto de instruções do Python ou por uma função ou método de uma de suas bibliotecas. Um dos objetivos da metodologia que usaremos é o de mesclar as várias fases da solução e agregar todas as informações na codificação e teste da solução. Também daremos enfase na funcionalidade, priorizando a descoberta do espaço de soluções e deixando mais para o final da disciplina a análise do comportamento dos programas para valores de entrada não esperados.

Ao iniciar a solução de um problema usando um computador, a primeira ação a ser feita é a leitura cuidadosa do problema para entender claramente o que se deseja resolver. Nem sempre isso é obtido com a primeira leitura. Releia o problema até que haja um entendimento claro da solução esperada. Uma vez definida a resposta desejada, temos que verificar que entrada será necessária para obter a solução do problema. O resultado dessa ação deve ser a descrição no cabeçalho do programa, como ilustrado na linhas 5-6 do Programa `perimetroareaQ.py`.

Após o entendimento do que se espera da solução, temos que descrever com mais precisão o domínio (conjunto de entrada) e imagem (conjunto de saída) da solução. Além disso, devemos ter uma ideia inicial de como e entrada gerará a saída desejada. Essas descrições detalham as especificações do estado inicial e final do programa. As especificações de entrada e saída para o problema do cálculo dos total de segundos de uma determinada área são escritas nas linhas que começam com

```
# pré:
```

e

```
# pós:
```

no programa `perimetroareaQ.py`.

Após a definição das pré e pós-condições, podemos descrever as principais variáveis que serão utilizadas no programa, bem como as bibliotecas do Python que serão utilizadas. Posteriormente, na medida em que o programa for sendo implementado, pode ocorrer a necessidade de utilizarmos outras variáveis e outras bibliotecas.

Temos agora que detalhar os passos necessários para transformar a entrada na saída desejada. Considerando que estamos resolvendo um problema usando um computador, cada um dos passos a serem executados pelo algoritmo/programa devem ser computacionalmente implementáveis, ou seja os passos devem ser possíveis de serem traduzidos por instruções da linguagem que está sendo utilizada, no caso Python. Descreveremos cada um dos passos do algoritmo/programa como um comentário no nosso programa.

Se olharmos o programa como uma função $f : \mathbb{E} \rightarrow \mathbb{S}$, onde $\mathbb{E}$ é a entrada (domínio) e $\mathbb{S}$ é a saída (imagem), o que temos que fazer é obter as subfunções $f_1, f_2, \ldots, f_n$ tal que $f = f_1 \circ f_2 \circ \ldots \circ f_n$. Além disso, cada uma das subfunções $f_i, 1 \leq i \leq n$ deve descrever uma instrução (ou conjunto de instruções) que possa ser implementada em Python (ou por alguma biblioteca). Para obtermos esse passo a passo da descrição do algoritmo usamos a técnica dos refinamentos sucessivos. Os três passos que compreendem a solução do problema do cálculo do número de segundos numa dada hora são documentados nas três linhas que começam com

```
# Passo
```

Os três passos que compreendem a solução do problema do cálculo do do perímetro e da área do quadrado são dados pelos Passos (algoritmo):

```
1 # Passo 1. Leia o lado do quadrado
2 # Passo 2. Calcule o perímetro e a área do quadrado
3 # Passo 2.1. Calcule o perímetro do quadrado
4 # Passo 2.2. Calcule a área do quadrado
5 # Passo 3. Imprima o lado, perímetro e área
```

Pode ocorrer que um dos passos do programa (f_i) já tenham sido desenvolvido anteriormente, nesse caso você pode adaptar a solução existente para ser reutilizada nesta solução do novo problema. Muitas vezes esta atividade irá requerer a combinação de duas ou mais funções para formar uma nova, ou ainda adaptar uma função para servir sua nova específica utilização.

Na medida que os refinamentos sucessivos forem chegando a um ponto em a tradução para uma instrução Python (ou de uma biblioteca), nada impede que já façamos a codificação. Pode ocorrer que a descrição envolva um conjunto de instruções e nesses casos, é recomendável só fazer a codificação após a finalização completa da subdivisão (refinamentos sucessivos). As linhas 14-15, 18, 20 e 22-24 no programa `perimetroareaQ.py` são exemplos de código `Python` necessário para resolver o problema o cálculo do perímetro e área de um quadrado..

Nas próximas aulas, usando essa metodologia que utilizaremos para projetar e implementar soluções computacionais, detalharemos mais cada uma das fases da metodologia. A seguir, descrevemos um programa na linguagem Python que implementa o algoritmo. Na medida que formos implementando as soluções em Python, descreveremos detalhes da

sintaxe da linguagem. O importante agora é entender os três passos principais do programa. A descrição do algoritmo, os passos, as variáveis utilizadas e as pré e pós-condições são incluídas no programa como comentários, precedidos `#` e servem para *documentar* o programa.

```
1  #-*- coding: utf-8 -*-
2  # Programa: perimetroareaQ.py
3  # Programador:
4  # Data: 26/03/2010
5  # Este programa lê o valor do lado de um quadrado e
6  # calcula e imprime o perímetro e a área do quadrado.
7  # início do módulo principal
8  # descrição das variáveis utilizadas
9  # float lado, perimetro, area
10
11 # pré: lado
12
13 # Passo 1. Leia o lado do quadrado
14 print('Entre com o valor do lado do quadrado: ')
15 lado = float(input()) # lê o fluxo e converte para float
16 # Passo 2. Calcule o perímetro e a área do quadrado
17 # Passo 2.1. Calcule o perímetro do quadrado
18 perimetro = 4 * lado
19 # Passo 2.2. Calcule a área do quadrado
20 area = lado * lado
21 # Passo 3. Imprima o lado, perímetro e área
22 print('Lado = {0:.2f}'.format(lado))
23 print('Perímetro = {0:.2f}'.format(perimetro))
24 print('Área = {0:.2f}'.format(area))
25
26 # pós: perimetro == 4 * lado and area == lado * lado
27 # fim do módulo principal
```

Posteriormente, iremos explicar com mais detalhes os significados de `pré` e `pós`. Aqui eles podem ser interpretados como sendo o domínio e a imagem, respectivamente, da função $y = Quadrado(x)$.

Vamos descrever algumas características do `Python` no programa acima. Como vimos anteriormente, a função `input()` pode ser utilizada para ler a entrada do programa. A instrução abaixo

```
# Passo 1. Leia o lado do quadrado
....
lado = float(input())
```

lê um número de ponto flutuante e armazena na variável `lado`. A função `input()` lê uma string da entrada padrão (no nosso caso o teclado). Quando usamos `float(input())`, é feito a conversão da string para um número de ponto flutuante. Após a conversão o valor de ponto flutuante é atribuído para a variável `lado`.

A função `print()` da instrução abaixo

```
# Passo 3. Imprima o lado, perímetro e área
print('Lado = {0:.2f}'.format(lado))
....
```

imprime a string `'Lado = {0:.2f}'` usando o formado especificado {`0:.2f`}. Como vimos anteriormente, o primeiro campo, 0 é associado a primeira variável descrita em `format(...)`, `lado` e o segundo campo, `.2f` indica o tipo (`f=float` e o formato como o número será exibido `.2f` indica que serão duas casas decimais após o `'.'`. Para o valor de `lado == 5.0`, na impressão haverá a "substituição" de {`0:.2f`} por 5.00. A "impressão" final dessa instrução fica:

```
Lado = 5.00
```

Usando um Editor ou o IDLE (ambiente do Python), edite o programa e salve num diretório de trabalho com o nome `perimetroareaQ.py`.

Verifique se você está no diretório onde o programa fonte `perimetroareaQ.py` foi salvo e compile o programa. Para interpretar/executar use o comando:

```
$ python perimetroareaQ.py
```

Os caminhos (`PATH`) para os módulos do Python devem estar devidamente definidos.

O arquivo `perimetroareaQ.py` indica o programa fonte que será usado na interpretação/execução.

Se você editou o programa corretamente a execução do comando de interpretação não gerará nenhuma advertência ou erro. Caso isso ocorra, veja o número da linha e verifique o que você digitou de forma errada.

Após ter corrigido todas as advertências e erros, execute o comando novamente. Se não houver mais nenhum erro, a execução do programa começará.

Teste o seu programa para diversos números tais como 5.0 e 10.0. Os exemplos abaixo ilustram o resultado da execução do programa para essas entradas.

Exemplo 1:

```
# formato da entrada
5.0

# formato da saída
Lado =  5.00
Perímetro = 20.00
Área = 25.00
```

Exemplo 2:

```
# formato da entrada
10.0

# formato da saída
Lado =  10.00
Perímetro = 40.00
Área = 100.00
10.0
```

Agora detalharemos um pouco mais a solução do problema de computar o total de segundos. Como já destacamos, a documentação dada pelos comentários (as linhas que são precedidas de `#`), pois auxilia na legibilidade e o resto do programa define os mecanismos computacionais para computar o `Total_Segundos`.

Dado uma medida de tempo em horas, minutos e segundos, como seriam os passos necessários para computar o total em segundos dessa medida de tempo. A entrada é a quantidade de horas, minutos e segundos e a saída é o total em segundos dado pela quantidade de horas multiplicada por 60×60 somado com a quantidade se minutos multiplicado por 60 e somado com a quantidade de segundos.

Após o entendimento do que se espera da solução, temos que descrever com mais precisão o domínio (conjunto de entrada) e imagem (conjunto de saída) da solução. Além disso, devemos ter uma ideia inicial de como e entrada gerará a saída desejada. Essas descrições detalham as especificações do estado inicial e final do programa. As especificações de entrada e saída para o problema do cálculo dos total de segundos de uma determinada área são escritas nas linhas que começam com

```
# pré:
```

e

```
# pós:
```

no programa `segundos.py`.

Após a definição das pré e pós-condições, podemos descrever as principais variáveis que serão utilizadas no programa, bem como as bibliotecas do Python que serão utilizadas. Posteriormente, na medida em que o programa for sendo implementado, pode ocorrer a necessidade de utilizarmos outras variáveis e outras bibliotecas.

Temos agora que detalhar os passos necessários para transformar a entrada na saída desejada. Considerando que estamos resolvendo um problema usando um computador, cada um dos passos a serem executados pelo algoritmo/programa devem ser computacionalmente implementáveis, ou seja os passos devem ser possíveis de serem traduzidos por instruções da linguagem que está sendo utilizada, no caso Python. Descreveremos cada um dos passos do algoritmo/programa como um comentário no nosso programa. Os três passos que compreendem a solução do problema do cálculo do número de segundos numa dada hora são documentados nas três linhas que começam com

```
# Passo
```

```
1 # Passo 1. Leia a entrada
2 # Passo 2. Calcule o total de segundos
3 # Passo 3. Imprima o total de segundos
```

no programa `segundos.py`. Sendo que o `Passo 1` pode ser refinado (subdividido) em:

```
1 # Passo 1. Leia a entrada
2 # Passo 1.1. Imprima uma mensagem informando a entrada
3 # Passo 1.2. Leia o número de horas
4 # Passo 1.3. Leia o número de minutos
5 # Passo 1.4. Leia o número de segundos
```

Pode ocorrer que um dos passos do programa (f_i) já tenham sido desenvolvido anteriormente, nesse caso você pode adaptar a solução existente para ser reutilizada nesta solução do novo problema. Muitas vezes esta atividade irá requerer a combinação de duas ou mais funções para formar uma nova, ou ainda adaptar uma função para servir sua nova específica utilização.

Abaixo descrevemos os detalhes do programa em `Python` que implementa esse algoritmo. Cada um desses passos está claramente anotado como um comentário no programa (`#`). Os comentários são úteis em programação como um dispositivo para descrever diretamente as ideias junto com o código. Neste nosso curso vamos usá-los com este propósito.

Na medida que os refinamentos sucessivos forem chegando a um ponto em a tradução para uma instrução Python (ou de uma biblioteca) , nada impede que já façamos a codificação. Pode ocorrer que a descrição envolva um conjunto de instruções e nesses casos, é recomendável só fazer a codificação após a finalização completa da subdivisão (refinamentos sucessivos). As linhas (17-20), 22 e 24 no programa `segundos.py` são exemplos de código `Python` necessário para resolver o problema o cálculo do números de segundos numa dada hora.

```
1  # -*- coding: utf-8 -*-
2  # Programa: segundos.py
3  # Programador:
4  # Data: 17/05/2016
5  # Este programa lê uma medida de tempo, dada em horas,
6  # minutos e segundos e calcula seu equivalente em segundos.
7  # início do módulo principal
8  # descrição das variáveis utilizadas
9  # int horas -> representa o valor da entrada em horas
10 # int minutos -> representa o valor da entrada em minutos
11 # int segundos -> representa o valor da entrada em segundos
12 # int totalSegundos -> representa o valor total em segundos
13
14 # pré: horas, minutos, segundos
15
16 # Passo 1. Leia a entrada
17 print('Entre com a quantidade de horas, minutos, segundos')
18 horas = int(input())
19 minutos = int(input())
20 segundos = int(input())
21 # Passo 2. Calcule o total de segundos
22 totalSegundos = 60*60*horas + 60*minutos + segundos
23 # Passo 3. Imprima o total de segundos
24 print('{0:d} Horas + {1:d} Minutos + {2:d} Segundos = {3:d} Segundos'. \
25                             format(horas, minutos, segundos, totalSegundos))
26
27 # pós: totalSegundos == 60*60*horas+60*minutos+segundos
28 # fim do módulo principal
```

O caractere \utilizado na função `print` do `Passo 3` indica que a função continua na próxima linha.

Após a finalização da codificação, temos que verificar se o programa não possui nenhum erro de sintaxe (instruções que o interpretador do Python não consegue executar). Uma vez

que o programa não tenha nenhum erro de sintaxe, temos que verificar de o programa não possui nenhum erro de semântica, ou seja, se ele gera as respostas esperada para todas as possíveis entradas. Executar o programa para todas as entradas possíveis não é factível na maioria dos casos. Existem várias abordagem para teste e verificação de programas. Nessa disciplina, inicialmente a validação priorizará a testagem do programa somente para os valores esperados da entrada. Posteriormente incorporaremos a avaliação do comportamento do programa para entradas não esperadas (p. ex. é fornecido como entrada uma palavra e a entrada esperada é um número inteiro). Para testar e verificar os programas, procuramos selecionar um conjunto de valores de entrada dentro de um conjunto de alternativas que explorem todas as variações possíveis permitidas pelas especificações da entrada do problema. Para cada execução, verificamos se a saída do programa satisfaz as especificações de saída. Como observamos acima, exitem várias técnicas para testar e avaliar a correção de um dado programa. Essas técnicas serão abordadas em outras disciplinas.

Se seguirmos esse roteiro, ao final teremos um programa com comentários que facilitam o entendimento de seu funcionamento e que facilitarão possíveis alterações futuras no programa. Sempre é bom destacar no cabeçalho a data e o programador que desenvolveu o programa. Durante a disciplina usaremos a metodologia acima para o desenvolvimento dos programas.

Nas próximas aulas, exploraremos com mais detalhes as fases do entendimento do problemas, entrada e saída e os refinamentos sucessivos, discutindo como desenvolver especificações as quais claramente definem o que necessitamos efetuar para solucionar o problema. Posteriormente, na medida que a complexidade dos problemas for aumentando, abordaremos com mais detalhes as demais fases, sempre com o intuito de assegurar que desenvolvimento do programa (software) está correto, utilizável e de fácil manutenção.

Vamos agora descrever uma forma de como ler várias variáveis do mesmo tipo no `Python`. Como vimos nos exemplos anteriores, a função `input()` pode ser utilizada para ler a entrada do programa. Podemos combiná-la com outras instruções. A instrução abaixo

```
# Passo 1. Leia a entrada
horas, minutos, segundos = map(int, input().split())
```

lê três números inteiros separados por espaços e armazena-os nas variáveis `horas`, `minutos` e `segundos`. A função `input()` lê uma string da entrada padrão (no nosso caso o teclado), o método `split()` retorna a lista de todas as palavras da string. A função `map`, com o argumento `int` "converte" cada palavra em um inteiro. Como o `Python` possibilita atribuição múltipla, os três inteiros são atribuídos às variáveis `horas`, `minutos` e `segundos`.

Caso a entrada seja a linha:

```
3 34 45
```

teremos `horas == 3`, `minutos == 34` e `segundos == 45`.

Usando um Editor ou o ambiente IDLE do Python, edite o programa e salve num diretório de trabalho com o nome `segundos.py`.

Verifique se você está no diretório onde o programa fonte `segundos.py` foi salvo e interprete/execute o programa. Para interpretar/executar use o comando:

```
$ python segundos.py
```

Os caminhos (`PATH`) para as classes devem estar devidamente definidos.

O arquivo `segundos.py` indica o programa fonte que será usado na interpretação/execução.

Se você editou o programa corretamente a execução do comando de interpretação não gerará nenhuma advertência ou erro. Caso isso ocorra, veja o número da linha e verifique o que você digitou de forma errada.

Após ter corrigido as advertências e erros, execute o programa. O interpretador inicia a execução do programa.

A **entrada** pode ser uma medida de tempo qualquer, dada em horas, minutos e segundos. Se os passos 1, 2 e 3 (**processo**) forem executados cuidadosamente a saída para a entradas dadas serão:

Exemplo 1:

```
# formato da entrada
3 34 45

# formato da saída
3 Horas + 34 Minutos + 45 Segundos = 12885 Segundos
```

Exemplo 2:

```
# formato da entrada
5 19 25

# formato da saída
5 Horas + 19 Minutos + 25 Segundos = 19165 Segundos
```

2 Desenvolvimento de Sistemas

Vimos o que é necessário para implementar um programa (Solução algorítmica do problema, Codificação, Interpretação e Execução). Na maioria das vezes o problema a ser resolvido envolve um conjunto de algoritmos e a codificação pode envolver um número muito grande de linhas de código. Veremos agora como podemos desenvolver um sistema utilizando técnicas de programação estruturada. Essa técnica possibilita um desenvolvimento eficiente, correto de programas e de fácil manutenção.

2.1 Ciclo de Vida do Desenvolvimento de um Sistema

Uma dada solução computacional para um problema pode envolver um grande equipe de projetistas e programadores. Nessas situações a implementação de um projeto de software é desenvolvido utilizando uma série de fases inter-relacionadas denominada **Ciclo de Vida do Desenvolvimento do Sistema**. Nos dias de hoje os conceitos de engenharia de software requerem uma abordagem sistemática e rigorosa para o desenvolvimento de software.

Um modelo conhecido é o **modelo da queda d´água**. No nosso exemplo, esse modelo consiste de seis fases (em alguns casos pode ser composto de cinco ou sete fases). A Figura 2 ilustra o modelo que iremos descrever.

O modelo da queda d´água começa com os **requisitos do sistema**. Nesta fase, o analista de sistemas define os requisitos que especificam o que o sistema proposto deve atender. A fase de **análise** trata das diferentes alternativas do ponto de vista do sistema, enquanto a

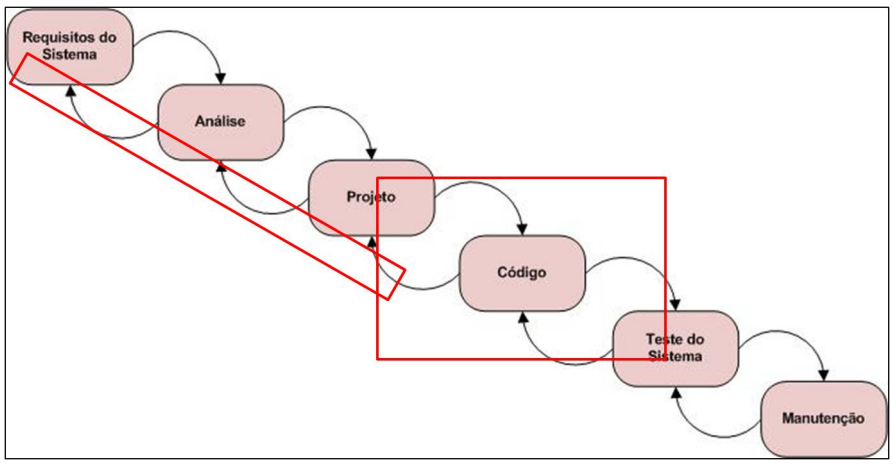

Figura 2: O Modelo Queda d´água.

fase do **projeto** determina como o sistema será construído. Na fase do projeto, as funções dos programas individuais que comporão o sistema são especificadas e o projeto dos arquivos e bancos de dados é finalizado. Na fase **código**, os programas são escritos (codificados). Esta é a fase que será tratada na disciplina de Algoritmos e Estrutura de Dados I. Depois que os programas foram escritos e testados por cada um dos programadores, o projeto segue para o **teste do sistema**. Todos os programas são testados juntos para garantir que o sistema funciona como um todo. A fase final de **manutenção** mantém o sistema em funcionamento depois que ele foi posto em produção. As setas no modelo da caixa d´água indicam que numa fase podem ser encontrados erros e omissões de outras fases anteriores, fazendo com que no desenvolvimento muitas vezes temos que voltar a fases anteriores para a correção desses erros e omissões. Nesta disciplina trabalharemos com problemas bem simples e concentraremos nossos estudos na *Solução Algorítmica* e *Desenvolvimento de Programas* que utilizem ideias preliminares das fases de requisitos, análise e projeto, codificação e ideias preliminares de testes (ficaremos restritos ao teste de programas).