



Resolução de Problemas com o Computador

Prof. Guilherme N. Ramos

A resolução de problemas utilizando um computador geralmente envolve três etapas:



1 Análise do Problema

É essencial que haja uma definição *clara* do que se deseja para que seja possível elaborar e implementar uma solução [1].

Entenda o problema. É tolice responder uma pergunta que você não entendeu, é preciso que esteja claro o que é *desconhecido* (o que se deseja obter?). Também é preciso saber o que é *conhecido*, para verificar se estes dados são suficientes para resolver a questão. Por exemplo, considere o problema *qual é a raiz quadrada de 1764?*

O desconhecido é o valor da raiz. São conhecidos o número em questão, que sua raiz é um número r tal que $r^2 = 1764$, e que há uma forma de calcular o valor de r . Estas informações são suficientes para resolver a questão, e há uma metodologia recomendada para isso [1].

Elabore um plano. Qual a ligação entre o conhecido e o desconhecido? Se não há uma ligação direta entre eles, há algum problema relacionado que pode ser útil na resolução? Se sim, este conhecimento é útil para o problema em questão? Pode ser necessário considerar problemas auxiliares se não houver conexão imediata. Já viu este problema antes? Ou algo parecido? É possível aproveitar algo conhecido neste caso? Se não, o problema não poderia ser apresentado de outra forma?

Por exemplo, considere o problema *qual é a raiz cúbica de 74088?* Não se sabe um método eficiente para computar a raiz cúbica de um número, mas sabe-se que a raiz é o número r tal que $r \cdot r^2 = 74088$. Então é possível adaptar o método de calcular a raiz quadrada para encontrar a raiz cúbica.

Execute o plano. Verifique cada passo do plano. Está claro que o passo está correto? É possível provar que o passo está correto? Esta análise visa garantir que a implementação dos passos planejados está de acordo com o esperado.

“As vezes a melhor forma de aprender algo é fazer errado e olhar para o que foi que você fez.”

Neil Gaiman

Examine a solução. É possível verificar o resultado? É possível obter o resultado de outra forma? É possível utilizar a solução ou o método em outro problema?

Por exemplo, suponha que se queira a soma de todos os números da sequência de números inteiros consecutivos de 0 a 100. O valor da soma destes números é o desconhecido, mas o intervalo é conhecido, $[0,100]$, assim como a execução de uma operação de soma. Então o conhecido é suficiente para resolver a questão. O plano é simples, definir o resultado como o primeiro número do intervalo, e ir somando a este cada um dos demais números. Cada passo executado pode ser demonstrado como correto. Após realizar as operações, a solução pode ser verificada (deveria ser 5050).

2 Projeto Do Algoritmo

“Antes de existirem computadores, já havia algoritmos. Mas agora que os computadores existem, há ainda mais algoritmos, e os algoritmos são a base da computação.”

Thomas Cormen

Há uma discussão interessante sobre a definição do conceito [formal] de algoritmo, mas as ideias convergem no pontos mais importantes. Segundo Knuth [2], um algoritmo tem as seguintes propriedades:

Finito: um algoritmo deve sempre terminar após um número finito de passos.

Bem definido: cada passo de um algoritmo deve ser precisamente definido, as ações a serem executadas devem ser especificadas de forma rigorosa e não ambígua para cada caso.

Entrada: quantidades que são dadas antes do início do algoritmo, originárias de um conjunto específico de objetos

Saída: quantidade que têm uma relação específica com a entrada.

Efetividade: todas as operações realizadas pelo algoritmo devem ser suficientemente básicas para que sejam executadas de forma exata e em tempo finito por um homem utilizando papel e caneta.

Já Harold Stone resume um algoritmo como um conjunto de regras que define, precisamente, uma sequencia de operações tais que cada regra seja efetiva e definitiva e que tal sequencia termine em tempo finito [3]. Por fim, David Berlinski o descreve como um procedimento finito, escrito em um vocabulário simbólico fixo, governado por instruções precisas, movendo-se em passos discretos, 1, 2, 3, ..., cuja execução não exige sagacidade, esperteza, intuição, inteligência ou perspicácia, e que mais cedo ou mais tarde chega ao fim. [4].

A realização da solução para o problema analisado é definida pelo projeto de do algoritmo, que pode ser visto como uma *sequência finita de instruções bem definidas e não ambíguas para executar uma tarefa*. Na prática, este deve considerar quem ou o que executará as instruções, no caso do computador, devem ser executáveis com uma quantidade finita de memória em um período de tempo finito.

Bolo simples

```
1  tigela ← ingredientes
2  bater tigela durante 5 min
3  untar forma
4  forma ← conteudo tigela
5  colocar forma no fogão
6  esperar 40 min
```

Soma 0 a 100

```
1  total ← 0
2  total ← total + 1
3  total ← total + 2
...
100 total ← total + 99
101 total ← total + 100
```

O algoritmo descreve um padrão de comportamento [5] utilizando instruções sequenciais, bifurcações e repetição. Este comportamento transforma os dados de entradas em dados de saída úteis (espera-se).

“Corretude é claramente a principal qualidade. Se um sistema não faz o que deveria fazer, então todas as outras coisas a seu respeito não têm importância.”

Bertrand Meyer

Corretude é um conceito ardiloso [6]. Por exemplo, um sistema GPS estaria correto se fornecesse a rota mais curta entre dois locais. Mas e se houver engarrafamento e, na verdade, o que você quer é a rota que o leve em menos tempo ao destino? Suponha que você tenha um algoritmo para aproximar o valor de π . Se o valor obtido for 3,1 está incorreto? A precisão necessária na aplicação do valor é o que determinaria a corretude (3,14 é suficiente para a maioria das pessoas, mas uma nave espacial provavelmente precisaria de mais casas decimais).

Ao projetar o algoritmo, muitas vezes começam a surgir ideias de como torná-lo mais rápido, mais genérico, mais “mais”. Antes de fazer isso, é preciso verificar se esta otimização (caso funcione) é realmente necessária? Certamente que não, pelo simples fato de não haver um programa pronto e testado que indique isso. Estas considerações quanto ao desempenho geralmente direcionam o projeto para código menos legível, mais complexo, e, possivelmente, incorreto. Melhorar o desempenho é uma boa ideia, desde que as alterações ocorram no momento certo (quando forem necessárias).

“Faça funcionar antes de fazer funcionar rápido.”

Bruce Whiteside

2.1 Representação de Algoritmos

Existem diversas formas de representar de um algoritmo: descrição narrativa, diagramas, pseudocódigo, linguagem de programação, linguagem de máquina, etc.; e não há consenso de qual é a melhor forma. Há, contudo, consenso que o algoritmo deve ser representado da forma mais clara possível, um código legível facilita o entendimento do programa, tornando-o mais claro e fácil de interpretar. Hoje em dia, a medida mais importante da qualidade de código é sua *legibilidade*¹.

Descrição Narrativa expressa o algoritmo em linguagem natural, a forma mais intuitiva de humanos se comunicarem. Por exemplo: *Bata as claras em neve. Reserve. Bata bem as gemas com a margarina e o açúcar. Acrescente o leite e farinha aos poucos sem parar de bater. Por último agregue as claras em neve e o fermento. Coloque em forma grande de furo central untada e enfarinhada. Asse em forno médio, pré-aquecido, por aproximadamente 40 min. Quando espetar um palito e sair limpo estará assado* [7].

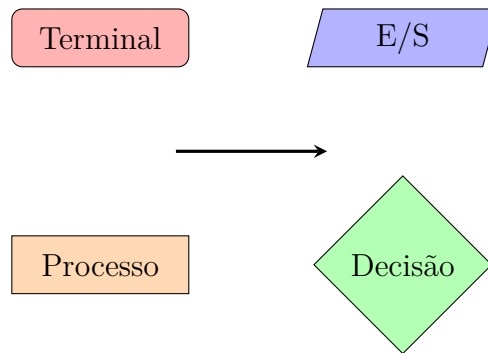
Parece simples, porque é a forma que estamos acostumados a usar para comunicação, mas é extremamente complexa pois considera a enorme capacidade humana de inferir informação incompleta. Para alguém que não esteja acostumado a atividades na cozinha, o algoritmo é ambíguo e impreciso.

“Se você não consegue escrever [a ideia] em Inglês, você não consegue codificar [a ideia].”

Peter Halpern

Fluxograma expressa o algoritmo graficamente com auxílio de formas geométricas.

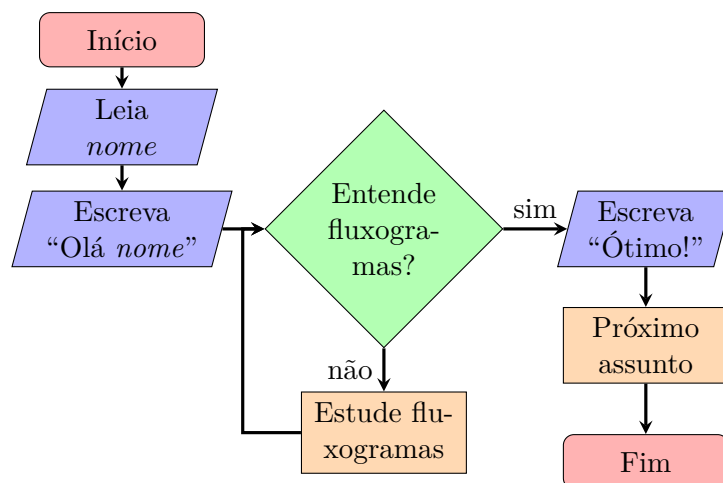
¹Há certos casos que não, mas no contexto de um curso de graduação, escreva código limpo.



Cada forma tem indica um tipo de ação, e nesta disciplina considera-se apenas as seguintes cinco formas:

- Terminal, que indica onde o algoritmo inicia e onde termina. Só pode haver um ponto de entrada, mas é possível haver múltiplos pontos de parada.
- Entrada/Saída, que indica a comunicação de dados entre o algoritmo e outros atores (usuário, sensor, etc.).
- Processo, que indica a execução de uma instrução/tarefa.
- Decisão, que indica uma bifurcação no fluxo (diversas possíveis ações subsequentes).
- Seta, que indica a direção do fluxo.

A utilização destas formas para representar a sequência de execução das instruções também é bastante intuitiva.



Pseudocódigo expressa o algoritmo em uma linguagem simples com estruturas de linguagem de programação, visando facilitar a compreensão humana (e não da máquina).

Exemplo de Pseudocódigo

```

1 Leia (nome)
2 Escreva ("Olá ", nome)
3 Se Entende (PSEUDOCÓDIGO) Então
4   Escreva ("Ótimo! Vamos para o próximo assunto.")
5 Senão
6   Escreva ("Vá estudar.")
7   Enquanto Não Entende (PSEUDOCÓDIGO)
8     Estude (PSEUDOCÓDIGO)
9   FimEnquanto
10 FimSe

```

Pseudocódigo tem uma linguagem um pouco mais flexível que uma linguagem de programação, podendo assim relaxar a rigidez de sintaxe, mas se assemelha na estrutura, facilitando a transcrição em código fonte. Além disso, geralmente se preocupa com detalhes que o fluxograma tende a ignorar, como declaração de tipos e variáveis. A construção do algoritmo geralmente envolve explicitamente as seguintes etapas:

“Pense em escrever pseudocódigo como se você fosse explicar a outra pessoa - geralmente não precisa ser conforme uma sintaxe específica desde que o que está acontecendo fique claro.”

Paul McMillan

1. Definição dos dados a serem manipulados:
 - Quais informações serão necessárias? (variáveis/constantes)
 - Quais os tipos de cada?
2. Obtenção destes dados (entradas).
3. Definição da sequência de execução dos processos.
4. Exibição dos resultados (saídas).

Por exemplo:

```
1  Algoritmo Le_2_Numeros_e_Mostra_Divisao
2  /* O nome indica claramente o que faz! */
3
4  Variáveis
5      numerador, denominador : real
6      /* Nomes indicam claramente para que servem! */
7
8  Início
9      Escreva ("Digite o numerador: ") /* Instruções claras. */
10     Leia (numerador)
11     Escreva ("Digite o denominador: ")
12     Leia (denominador)
13     Escreva ("A divisão é: ", numerador / denominador)
14  Fim
```

Os dados são manipulados na memória em função de seus endereços, mas é muito inconveniente (para o humano) ficar lidando com rótulos como 0xA23FCB00 (que a máquina entende). Desta forma, utiliza-se *identificadores* como nomes (rótulos) que identificam variáveis, funções, etc. na construção de um algoritmo computacional. É bem mais fácil lidar com numerador ou Escreva que com 0x7CABECA5 e afins.

Entretanto, certas palavras têm significados específicos no contexto, e seus identificadores e são chamadas *palavras reservadas*. Desta forma, quem elabora o algoritmo não pode usá-las indiscriminadamente. Alguns exemplos (não exaustivos) de palavras reservadas:

- | | | |
|-------------------------|-------------------|-------------------------|
| 1. Algoritmo | 4. Função | 7. Não |
| 2. Início/Fim | 5. Retorne | 8. Para/Até/FimPara |
| 3. Constantes/Variáveis | 6. Se/Senão/FimSe | 9. Enquanto/FimEnquanto |

Geralmente os identificadores têm algumas regras para serem formados. Por exemplo, há distinção entre ‘A’ e ‘a’, o primeiro caractere não pode ser numérico, caracteres especiais (pontuação, de controle, etc.) não são aceitos, e outras normas arbitrárias.

Válidos	Inválidos
nome	3CD
Entende	minha idade
NOTA2	DA*TA
prova_1	media.final

A implementação de um programa exige que a memória seja manipulada corretamente, portanto é necessário saber o que está armazenado e onde, de modo que o processo seja realizado corretamente. É preciso separar (e identificar) cada bloco da memória necessário, e definir seu tipo (que determina como lidar com o bloco e o que pode ser feito com o dado).

A declaração de uma *constante* identifica um espaço na memória do computador que é reservado para execução do algoritmo e utilizado para armazenar um determinado tipo de dado cujo valor não pode ser alterado ao longo da execução (daí o nome *constante*). Uma constante é extremamente útil para centralizar uma informação imutável (durante a execução).

Este conceito contrasta com a declaração de uma *variável*, que identifica um espaço na memória do computador que é reservado para execução do algoritmo e utilizado para armazenar um determinado tipo de dado cujo valor pode ser alterado ao longo da execução. É irrelevante se o valor é alterado ou não, o importante é que, caso desejado, é possível alterá-lo. Uma variável é útil para armazenar resultados que são obtidos ao longo da execução.

```
1  Algoritmo PesoDaTurma
2  Constantes
3      g : real ← 9.78 // m/s2
4  Variáveis
5      massa, acumulado : real
6      nome : string
7  Início
8      acumulado ← 0
9      Para Cada aluno ∈ turma
10         Escreva ("Digite seu nome: ")
11         Leia (nome)
12         Escreva ("Digite sua massa: ")
13         Leia (massa)
14         Escreva (nome, " seu peso é ", massa*g)
15         acumulado ← acumulado + massa
16     FimPara
17     Escreva ("O peso da turma é ", acumulado*g)
18 Fim
```

A constante g define o valor da gravidade na Terra, mas este mesmo procedimento poderia ser utilizado para calcular o peso de uma turma de marcianos, bastaria trocar o valor da constante para 3,711 (gravidade em Marte). Já a variável *acumulado* acumula a massa de cada aluno (entrada de dados), e ao final é utilizada para calcular o valor do peso da turma.

A utilização de variáveis em algoritmos é tamanha que merece algumas sugestões:

1. uma variável deve ter uma única utilidade;
2. uma variável não deve levar um significado de um domínio para outro; e
3. uma variável não deve ter mais de um significado ao mesmo tempo.

Variáveis e constantes armazenam dados, que são classificados dentre os diversos tipos possíveis. O tipo determina:

- quais são os valores possíveis para o dado;
- quais são as operações possíveis de serem realizadas com o dado;
- o significado do dado; e
- como o dado pode ser armazenado.

Os tipos mais comuns são: numérico (inteiros, reais), lógico (booleano), e simbólico. E as operações que podem ser realizadas sobre os tipos definem a realização de um procedimento que gera um novo valor a partir de valor(es) de entrada. Para tanto, utiliza-se um *operador*, que define a ação/procedimento a ser realizada, e um ou mais *operandos*, o(s) valor(es) envolvido(s) na operação.

As operações numéricas são mais familiares. Se *atribuo* o valor 1 a variável x ($x \leftarrow 1$), é direta a implicação sobre seu valor após a operação. Assim como calcular a soma de duas variáveis ($x + y$) ou atribuir o resultado desta operação a outra variável ($z \leftarrow x + y$). Já operações com lógica booleana podem não ser tão familiares, mas também são bastante intuitivas. Um valor que *não é falso* (F) tem de ser *verdadeiro* (V); a comparação ($x < y$) entre dois números só pode ter um dos dois Valores lógicos (V/F).

Ao executar operações, é preciso definir a ordem em que cada operação será realizada. Esta noção de *precedência* faz diferença no resultado conforme cada implementação do interpretador. Por exemplo, $1 + 2 \times 3$ resulta em 9 ou em 7? Depende da precedência dos operadores $+$ e \times (o mais comum é que a multiplicação tenha precedência, portanto o resultado seria 7). Em alguns casos, a indefinição de precedência pode levar a *comportamento não-definido* (o que implica que não se pode garantir o padrão de comportamento do algoritmo).

Na dúvida, você pode garantir o resultado utilizando parênteses para explicitar a ordem das operações.

$$(1 + 2) - (3 * (20/5)) + (2^3) = -1$$

3 Implementação

“Conhecimento não tem valor a não ser que seja posto em prática.”

Anton Chekhov

Um algoritmo computacional é um algoritmo a ser executado por um computador, ou seja, uma *sequência de instruções que vai manipular dados*. As instruções são comandos que determinam a forma pela qual os dados devem ser tratados, e os dados são informações recolhidas/fornecidas por diversos meios e que serão processadas pelas das instruções.

“O propósito de uma linguagem de programação é permitir que você expresse suas ideias em código.”

Bjarne Stroustrup

Você só é capaz de realmente programar uma solução para a tarefa quando a entende

3.1 Codificação

Codificação é a escrita do algoritmo em uma *linguagem de programação*, de modo que as instruções possam ser interpretadas pela máquina. Linguagem de programação:

Exemplo de código em linguagem C

```
1 scanf("%s", nome);
2 printf("Olá %s.\n", nome);
3
4 if(entende(LINGUAGEM_C))
5     printf("Ótimo! Vamos para o próximo assunto.\n");
6 else {
7     printf("Vá estudar.\n");
8     do
9         estude(LINGUAGEM_C);
10    while(!entende(LINGUAGEM_C));
11 }
```

“Programação de qualidade não é aprendida por generalizações, mas ao ver como programas significativos podem ser feitos de forma limpa, legível, de fácil manutenção e alteração, voltado para humanos, eficiente, e confiável, pela aplicação de bom senso e boas práticas de programação. O estudo cuidadoso e a imitação de bons programas leva a melhor codificação.”

Peter Norvig

Após a escrita do código, geralmente há:

Compilação: “tradução” das instruções em uma linguagem de programação para outra linguagem.

Execução: decodificação e execução das instruções do programa.

Verificação: testar a execução do programa com uma grande variedade de dados de entrada.

Depuração: Encontrar e corrigir erros: de compilação, de execução, e de lógica de programação.

Documentação: descrição dos passos da solução, do método utilizado, etc.

Manutenção: alterações no programa.

As duas últimas atividades são de muita importância, mas fora do escopo desta disciplina.

Pode parecer contraditório, mas a linguagem de programação tende a ser mais um detalhe na resolução de um problema. Se você entender como um computador funciona e como elaborar algoritmos computacionais corretos, tem as principais ferramentas para implementar uma solução (em código). A etapa de entender o problema e propor uma solução adequada é a mais importante na resolução, a linguagem é apenas a forma de representar o algoritmo (que pode facilitar muito a sua vida).

Não existe a “melhor” linguagem de programação (embora C tenha os melhores desempenho e retorno sobre investimento). Cada uma tem suas próprias características, vantagens e desvantagens. Idealmente, você deve aprender algumas linguagens para entender melhor as diferenças entre elas e poder identificar qual seria a mais indicada para o problema em questão. Por exemplo [8]:

- abstração de classes (C++/Java);
- abstração de funções (Lisp/Haskell);
- declaração de especificações (Prolog/C++);
- abstração sintática (Lisp);
- paralelismo (Clojure/Go).

Minha sugestão é que você aprenda, nesta ordem:

1. C (para entender como as coisas realmente funcionam);
2. Python (provavelmente a linguagem de maior “aplicabilidade” do momento - embora R seja a recomendada para análise estatística);
3. Haskell (mágica!).

Hoje em dia já se pode “programar” sem conhecer uma linguagem de programação. Bom, pelo menos brincar com os principais conceitos: light-Bot, e The Codex of Alchemical Engineering. E quando já tiver um pouco mais de experiência: Code Combat, Code Hunt, Elevator Saga, CodinGame, Robocode e URI.

E por falar em recursos on-line, existem excelentes oportunidades para ocupar aquele espaço entre 00:00 e 06:00 horas: Coursera, OpenCourseWare, Udacity, e edX.

3.2 Linguagem de Programação C

Em 12/2014, a linguagem de programação C era a mais popular no mundo [9] (e tem sido por muito tempo), provavelmente por [pelo menos] três boas razões: desempenho, ubiquidade, e portabilidade.

Além disso, é uma linguagem pequena e que produz código fácil de se ler e entender. No contexto de aulas introdutórias de computação, há mais uma enorme “vantagem”: é relativamente próxima a linguagem de máquina, portanto você sabe como as coisas funcionam no nível mais detalhado.

“O espírito da linguagem C” descreve como a linguagem visa proporcionar ao programador a melhor ferramenta possível, com as seguintes diretrizes:

- Confie no programador.
- Não impeça o programador de fazer o que precisa ser feito.
- Mantenha a linguagem [de programação] pequena e simples.

Como toda linguagem, C tem uma lista de palavras reservadas e também regras para formação de identificadores:

1. Palavras reservadas não podem ser usadas.
2. Há distinção entre ‘A’ e ‘a’.
3. O primeiro caractere não pode ser numérico.
4. Caracteres especiais (pontuação, de controle, etc.) não são aceitos.
5. Apenas os 31 primeiros caracteres são considerados para variáveis externas.

Válidos	Inválidos
nome	3CD
Entende	minha idade
NOTA2	DA*TA
prova_1	media.final

O tipo do dado faz toda a diferença nas operações que podem ser realizadas (veja 02_Operadores) e nos valores que podem ser armazenados (veja o arquivo 02-limites.c). Mas é interessante destacar que os limites e a precisão dos valores dependem da arquitetura do computador e que o valor lógico *falso* é equivalente ao valor numérico 0 (zero), e como *verdadeiro* é *não falso*, qualquer número diferente de zero é considerado.

Também há a questão de precedência, que pode levar a comportamento não-definido. Na dúvida, *sempre use parênteses para explicitar a ordem das operações*.

3.2.1 Código

Todo programa também tem uma estrutura inicial definida, que pode ser resumida da seguinte maneira:

```
1  /* Cabeçalho          */
2  /* Bibliotecas        */
3  /* Ponto de Entrada   */
```

O *cabeçalho* deve identificar o arquivo, contendo informações importantes. Em tempos de versionamento de código, isso provavelmente pode ser limitado a licença/direitos autorais e comentários exigidos por ferramentas de documentação automática. Se for o caso, o cabeçalho pode também contextualizar a situação em que se faz necessário o código.

As *bibliotecas* oferecem recursos (comportamentos) que podem ser utilizados por meio de interfaces bem definidas. Isso implica que não é preciso implementar novamente certos comportamentos, basta reutilizar o código da biblioteca (possivelmente em projetos independentes). Por exemplo, uma calculadora pode *imprimir* o resultado de uma soma, assim como um programa de busca pode *imprimir* o resultado de uma pesquisa.

O *ponto de entrada* indica onde a execução é iniciada, o começo da implementação de seu algoritmo. Em C, este ponto é a função `main` (veja o arquivo 01-hello_world_detalhado.c).

Há muitos recursos disponíveis sobre a linguagem C (por exemplo: C-FAQ ou The Basics of C Programming), mas a única forma de realmente aprender a programar é estudando (exemplos) e implementando código (exercícios).

Por fim, duas questões de estilo. Nesta disciplina será considerado o padrão ANSI C, definido para a linguagem pelo American National Standards Institute em 1989 e que garante a portabilidade dos programas gerados, independente do sistema operacional. Será considerado também o uso de comentários em código, para forçar o aluno a descrever de forma clara e breve o que seu código está fazendo (ou pelo menos indicar o que se gostaria que fosse feito).

3.3 Linguagem de Programação Python

No contexto desta disciplina, há código em linguagem Python para cada implementação em linguagem C. O ideal é se acostumar com ambas, mas há uma “tendência” no CIC de se considerar

que *todos os alunos sabem programar em linguagem C*. A linguagem Python é uma linguagem de programação que permite que você trabalhe rapidamente e integre sistemas mais eficientemente². A linguagem foi projetada com ênfase no esforço do programador sobre o esforço computacional, priorizando a legibilidade do código sobre a velocidade ou expressividade.

Além das inúmeras aplicações e da crescente popularidade da linguagem [9], a maior “vantagem” é a facilidade com que pode ser rapidamente utilizada no contexto de aulas introdutórias de computação. Como o “espírito da linguagem C”, há o Zen de Python (com exemplos).

Como toda linguagem, Python tem uma lista de palavras reservadas e também “sugestões” para formação de identificadores: o PEP 8.

Como na linguagem C, o tipo do dado faz toda a diferença nas que podem ser realizadas (veja 02_Operadores) e nos valores que podem ser armazenados (veja o arquivo 02-limites.py). Mas é interessante destacar que os limites e a precisão dos valores dependem da arquitetura do computador e que o valor lógico *falso* é equivalente ao valor numérico 0 (zero), e como *verdadeiro* é *não falso*, qualquer número diferente de zero é considerado.

Também há a questão de precedência, que pode levar a comportamento não-definido. Na dúvida, *sempre use parênteses para explicitar a ordem das operações*. No contexto de introdução a programação, e comparando a linguagem C, a maior diferença será o operador de divisão (`//` - veja 03-divisao.py).

3.3.1 Código

Todo programa também tem uma estrutura inicial definida, que pode ser resumida da seguinte maneira:

```
1 # Cabeçalho
2 # Módulos
3 # Ponto de Entrada
```

Os três elementos têm utilidade como na linguagem C, “*módulo*” é o termo em linguagem Python para designar um biblioteca (veja o arquivo 01-hello_world_detalhado.py).

Há muitos recursos disponíveis sobre a linguagem Python (por exemplo: Python FAQ), mas a única forma de realmente aprender a programar é estudando (exemplos) e implementando código (exercícios).

Por fim, duas questões de estilo. Nesta disciplina será considerado o padrão PEP 8, definido pelo criador da linguagem. Será considerado também o uso de comentários em código, para forçar o aluno a descrever de forma clara e breve o que seu código está fazendo (ou pelo menos indicar o que se gostaria que fosse feito).

4 Comentários

Comentários são trechos de um arquivo de programa que são ignorados pelo compilador/interpretador, podem ser utilizados livremente para tornar um programa mais fácil de se entender [10]. Comentários são, no melhor caso, um mal necessário pois servem para tentar compensar a falha do programador ao se expressar em código [11]. Os melhores comentários ajudam o entendimento do programa ao indicar, brevemente, algum detalhe importante ou fornecer uma ideia dos procedimentos como um todo [12]. Quando possível, escreva código ao invés de comentários [13].

“Não comente código ruim — reescreva-o.”

Brian Kernighan & P. J. PLaugher

Um comentário deveria explicar o *porquê*, não o *como*, de um trecho de código. Alguns tipos de comentários podem bastante úteis. Por exemplo, o cabeçalho de um programa dando detalhes

²<https://www.python.org/>

do projeto, do(s) autor(es), das condições ou licença de uso, ou indicativos de funcionalidades que devem ser incluídas (num futuro próximo).

Outros tipos podem dificultar a leitura ou entendimento do código. Algo que não faça sentido unto ao trecho de código, algo redundante com o código,

Comentários “bons”

```
1 float InvSqrt(float x) {
2 /* Calcula o inverso da raiz de forma
3 eficiente, usando o método de Newton-
4 Raphson com valor inicial otimizado. */
5
6 float xhalf = 0.5f*x;
7 int i = 0x5f3759df - ((*(&int*)&x)>>1);
8 x = (*(&float*)&i)*(1.5f - xhalf*x*x);
9 return x;
10 }
11
12 /* TO-DO: Incluir exemplo de licença. */
```

Comentários “ruins”

```
1 # Armazena o dia do mes.
2 int dia_do_mes;
3
4 # Calcula a parte fracionária de n.
5 r = InvSqrt(n)
6
7 # Código de Fulano
8 if batata == 1: # batata ???
9     imprime_pagina()
10    confirma_impressao()
11 # if
```

“Se você vai incluir um comentário, use o tempo necessário para garantir que será o melhor que pode escrever.”

Robert C. Martin

Referências

- [1] George Pólya. *How to solve it: a new aspect of mathematical method*. Princeton science library. Princeton University Press, Princeton [N.J.], expanded princeton science library ed edition, 2004.
- [2] Donald Ervin Knuth. *The Art of Computer Programming 1. Fundamental Algorithms*. Addison Wesley, 1997.
- [3] Harold S. Stone. *Introduction to Computer Organization and Data Structures*. McGraw-Hill, Inc., New York, NY, USA, 1971.
- [4] David Berlinski. *The Advent of the Algorithm: The 300-Year Journey from an Idea to the Computer*. AUDIBLE STUDIOS ON BRILLIANCE, 2016.
- [5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.
- [6] Thomas H. Cormen. *Algorithms unlocked*. The MIT Press, Cambridge, Massachusetts, 2013.
- [7] Maria F. N. Vechi. Bolo Simples. <http://www.tudogostoso.com.br/receita/29124-bolo-simples.html>.
- [8] Peter Norvig. Aprenda a programar em dez anos, 2001. <https://pihisall.wordpress.com/2007/03/15/aprenda-a-programar-em-dez-anos/>.
- [9] TIOBE Index. <http://www.tiobe.com/tiobe-index/>.
- [10] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.
- [11] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

- [12] Rob Pike Brian W. Kernighan. *The Practice of Programming*. Addison Wesley, 1999.
- [13] Herb Sutter and Andrei Alexandrescu. *C++ coding standards: 101 rules, guidelines, and best practices*. The C++ in-depth series. Addison-Wesley, Boston, 2005.