Algoritmos e Programação I: Aula 04*

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

Sumário

1 Projetando e Implementado Soluções Computacionais

A principal razão das pessoas apreenderem linguagens de programação é para utilizar o computador como uma ferramenta para solução de problemas. Vimos na aula anterior que o uso de uma metologia é fundamental para o projeto e implementação e soluções computacionais. Baseado no MAPS, podemos usar as seguintes fases nessa tarefa:

1. Fase 1 - Descrição Detalhada (entendimento do problema).
2. Fase 2 - Domínio e Imagem do problema (Especificações de entrada e saída).
3. Fase 3 - Algoritmo (Refinamentos Sucessivos - Passos do Algoritmo).
4. Fase 4 - Codificação e Documentação (Finalização).
5. Fase 5 - Teste e Verificação do Programa.

*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.

A utilização dessa metodologia no processo de solução de problemas algorítmicos com o auxílio de computador pode ser detalhada da seguinte maneira:

- As fases 1 e 2 são as responsáveis pela análise e especificação do problema.
- A fase 3 é a responsável pelo desenvolvimento do algoritmo.
- A fase 4 é a responsável pela codificação e documentação do programa numa linguagem de programação (*Python* na nossa disciplina).
- A fase 5 é a responsável pelo teste e verificação do programa.

No modelo de ciclo de vida da Queda d´água, além das fases propostas pela metodologia tem a fase da *manutenção*. Essa fase é muito importante mas não será abordada nesta nossa disciplina. Ela é utilizada em sistemas mais complexos.

A forma como abordaremos as fases se *Requisitos do Sistema*, *Análise*, *Projeto* e *Teste do Sistema* será adaptada ao desenvolvimento de programas dentro do escopo da disciplina de Algoritmos e Programação I.

2 Usando uma Metodologia na Solução de Problemas

Vamos utilizar a metodologia descrita anteriormente para solucionar um problema usando um computador. As duas primeiras fases são a descrição do entendimento do problema e o domínio e imagem do problema (especificações - entrada e saída). A fase 3 descreve a função que transforma a entrada (domínio) na saída (imagem). Essa função tem que ser descrita de acordo com as funcionalidades do computador. Para isso, utilizaremos um procedimento (algoritmo) descrevendo os passos necessários (de acordo com as funcionalidades do computador) para obter a saída desejada a partir dos dados de entrada para o problema. Visto que o computador é uma máquina que não possui nenhuma capacidade inerente para solução de problemas, este procedimento deve ser formulado como uma sequência detalhada de passos simples. Essa sequência de passos simples descreve o algoritmo para solucionar o problema a ser resolvido.

Como vimos na aula anterior, os passos que compõe um algoritmo devem ser organizados em uma maneira lógica e clara tal que o programa que implementa este algoritmo é similarmente bem estruturado. Além disso os passos dos algoritmos possuem três métodos de controle:

1. **Sequencial:** Os passos são executados em uma maneira estritamente sequencial, cada passo sendo executado exatamente uma vez.
2. **Seleção:** Uma de diversas alternativas de ação é selecionada e executada.
3. **Repetição:** Um ou mais passos é executado repetidamente.

Estas três estruturas parecem ser muito simples, mas na realidade elas são suficientemente poderosas que qualquer algoritmo pode ser construído usando essas estruturas.

Programas para implementar algoritmos devem ser escritos em uma linguagem que possa ser entendida pelo computador. Portanto, descreveremos os algoritmos usando "passos" que possam ser implementados por uma linguagem de programação de computadores. Os algoritmos também podem descritos com o uso de *Diagrama de Estruturas* e *Fluxogramas* para auxiliar no Desenvolvimento de Programas.

Usaremos a sintaxe da linguagem de programação `Python` para implementar os passos dos algoritmos:

1. Os símbolos usuais são usados no computador para operações aritméticas: + para a adição, − para a subtração, ∗ para a multiplicação e // para a divisão inteira e / para divisão com ponto flutuante.

2. Nomes simbólicos (identificadores) são usados para representar as quantidades sendo processadas pelo algoritmo.

3. A utilização de comentários. Isto é feito usualmente iniciando cada linha de comentário com o símbolo `#`.

4. Funções definidas no `Pyrthon`, tais como `input()` para indicar uma operação de entrada; `print()` para operações de saída.

5. Indentação é usada para agrupar blocos de instruções.

Vamos desenvolver soluções computacionais para problemas que utilizem a estrutura sequencial usando a metodologia acima. Nas próximas aulas descreveremos soluções de problemas, cujos algoritmos, utilizam as estruturas de controle de seleção e de repetição.

2.1 Análise do Problema e Especificação

Como vimos anteriormente, a descrição inicial de um problema pode ser um tanto vaga e imprecisa, a primeiro fase no processo de solução de um problema é rever o problema cuidadosamente (*Entendimento do Problema*) a fim de obter uma ótima compreensão do problema. Após essa fase temos que determinar (*As Especificações - Domínio e Imagem*) sua **entrada** (**input**) - quais informações são dadas e quais são importantes na solução do problema - e sua **saída** (**output**) - quais informações devem ser produzidas para solucionar o problema (**pré** e **pós-condições**). Entrada e Saída são as duas partes principais na *especificação* do problema e para os problemas que serão apresentados nesta disciplina, usualmente elas não são difíceis de serem identificadas. Contudo, nos problemas do mundo real encontrados por um programador profissional, as especificações do problema muitas vezes incluem outros itens e um esforço considerável pode ser necessário para formular completamente o problema. Com a utilização de exemplos, vamos ilustrar as duas primeiras fases (*Entendimento do Problema e as Especificações*) da metodologia.

Problema 1: Calculando Receita. Epaminondas Furtado instala cabo coaxial para a Companhia de TV a Cabo Tuiuiu. Para cada instalação, existe uma taxa de um serviço básico de R$ 125 e uma taxa adicional de R$ 2 por cada metro de cabo. Durante o mês de Janeiro, Epaminondas instalou 1263 metros de cabo em 45 diferentes locais. Qual foi o total de receita gerada por ele para esse mês?

Para esse problema, praticamente a própria definição do problema fornece um entendimento adequado do problema.

```
Este algoritmo recebe como entrada o número de instalações e o preço da
tarifa básica de instalação do ponto de TV a cabo, o total de metros de
cabo coaxial utilizados nas instalações e o custo de cada metro de cabo
instalado por um dado instalador de TV a cabo. Calcula a receita gerada
pelo instalador e imprime o resultado.
```

Vamos agora efetuar as Especificações - Domínio e Imagem (pré e pós-condições - entrada e saída). Identificar o entrada e saída neste problema é fácil:

Entrada	**Saída**
Tarifa serviço: R$ 125.00	Receita Gerada =
Custo metro de cabo: R$ 2.00	45*125.00 +
Número de instalações: 45	1263*2.00 =
Metros de Cabo: 1263	R$ 8151.00

Os outros itens de informação - nome do empregado, seu trabalho, o nome da companhia, e o mês - não são relevantes (pelo menos para esse problema) e podem ser ignorados.

Determinar a receita gerada por Epaminondas Furtado pode ser feita facilmente de forma manual, ou mais facilmente ainda usando uma calculadora, e não justificaria o desenvolvimento de um programa para sua solução. Um programa escrito para resolver esse problema em particular seria utilizado apenas uma vez; se Epaminondas instalar cabos em 42 locais em Fevereiro usando 1455 metros de cabo ou se houver mudança no preço da taxa se serviço básico ou custo por metro de cabo, teríamos um novo problema requerendo o desenvolvimento de um novo programa. Isto é obviamente perda de esforço, visto que cada problema desse é um caso especial de um problema mais geral de encontrar a receita gerada por qualquer número de instalações, qualquer quantidade de cabos, qualquer taxa de serviço básico, e qualquer custo de unidade de cabo. Um programa que soluciona o problema geral pode ser utilizado em uma variedade de situações e é consequentemente mais útil que um projetado para resolver somente o caso especial do problema original.

Portanto, um aspecto importante do estágio do entendimento do problema (análise do problema) é a *generalização*. O esforço envolvido nas fases posteriores do processo de solução de problemas exige que o programa eventualmente desenvolvido seja suficientemente flexível, que ele não solucione somente o problema específico mas também qualquer problema relacionado do mesmo tipo com pouca, se alguma, modificação requerida. Portanto, neste exemplo as especificações do problema podem ser melhor formuladas em termos mais gerais:

Entrada	**Saída**
Tarifa serviço	Receita Gerada
Custo metro de cabo	Tarifa serviço * Número de instalações +
Número de instalações	Custo metro de cabo * Metros de cabo
Metros de Cabo	

Usamos variáveis do tipo `float` para armazenar as informações da Tarifa de serviço (`tarifaServico`), Custo metro de cabo (`custoMetro`) e Receita Gerada (`receita`), e variáveis do tipo `int` para armazenar o Número de instalações (`instalacoes`) e Metros de Cabo (`metrosdeCabo`). Nesse caso, temos a função

```
receita(instalacoes, tarifaServico, metrosdeCabo, custoMetro) =
                  instalacoes * tarifaServico + metrosdeCabo * custoMetro
```

e as pré condições (domínio da função) e pós-condições (imagem) são as seguintes:

```
# pré: tarifaServico custoMetro instalacoes metroDeCabo
```

```
# pós: receita == instalacoes * tarifaServico + metrosdeCabo * custoMetro
```

Obviamente, poderíamos generalizar ainda mais - permitir diversos funcionários, outros tipos de taxas, diferentes tipos de cabos, e assim por diante - mas devemos estabelecer um limite em algum ponto ou iríamos generalizando para sempre. Nesta introdução elementar para o processo de solução de problemas, os nossos exemplos serão bastante simples.

2.2 Passos do Algoritmo - Refinamentos Sucessivos

Como observamos anteriormente, a entrada para este problema consiste do número de instalações, da tarifa básica, do número de metros usados e do custo por cada metro de cabo, e a saída a ser produzida é o valor da receita gerada.

O primeiro passo em um algoritmo para solucionar este problema é o de obter os valores para os itens da entrada (input) - tarifa básica do serviço, custo por metro de cabo e número de instalações. A receita gerada pode então ser obtida pela multiplicação do número de instalações pela tarifa básica, pela multiplicação do custo de metro de cabo pelo número de unidades de cabo, e somando estes dois produtos. Finalmente, o valor do saída (output) - receita gerada - deve ser impressa.

A cada etapa, vamos incorporando as informações no código do programa. Com isso, este algoritmo pode ser expresso como segue:

```
# Algoritmo: receita
# Programador:
# Data: 08/03/2011
# Este algoritmo recebe como entrada o número de instalações e o preço da
# tarifa básica de instalação do ponto de TV a cabo, o total de metros de
# cabo coaxial utilizados nas instalações e o custo de cada metro de cabo
# instalado por um dado instalador de TV a cabo. Calcula a receita gerada
# pelo instalador e imprime o resultado.
# início do algoritmo
# descrição das variáveis locais utilizadas
# int   instalacoes, metrosdeCabo
# float tarifaServico, custoMetro, receita

# pré: instalacoes tarifaServico metrodeCabo custoMetro

# Passo 1. Leia os dados de entrada
# Passo 2. Calcule a receita gerada
# Passo 3. Imprima a receita gerada

# pós: receita == instalacoes * tarifaServico + custoMetro * metrosDeCabo
# fim algoritmo
```

Este algoritmo usa somente controle sequencial; os passos são executados em ordem, do início ao final, cada passo sendo executado somente uma vez. Contudo, para outros problemas, a solução pode requerer que alguns dos passos sejam somente executados em determinados situações e ignorados em outras. Além disso, temos situações onde um passo (ou conjunto de passos) deve ser executado mais de uma vez. Esses tipos de algoritmos serão abordados nas próximas aulas.

2.3 Finalização da Codificação e Documentação

A fase 4 no desenvolvimento da solução computacional para um problema algorítmico é a codificação e a documentação. Para efetuar a codificação, devemos escolher uma linguagem de programação. Vamos codificar o algoritmo do cálculo da receita (`receita`) usando a linguagem `Python` (`receita.py`).

Na descrição do Algoritmo para o Cálculo da Receita (Fases 1 a 3) , foi apresentada uma solução detalhando os passos do algoritmo. Temos que descrever essa solução usando as funcionalidades do computador usando a sintaxe da linguagem `Python`.

Como vimos anteriormente, em `Python`, as variáveis podem ser de vários tipos (`int`, `float`, etc.) e como vimos na aula prática os nomes das variáveis possuem regras para serem utilizados, além disso, as variáveis não podem usar palavras reservadas da linguagem (`while`, `if`, etc.).

O Python é uma linguagem multi-paradigma e utiliza tipos dinâmicos, em função disso, na implementação do não é necessário declarar as variáveis. Como uma estratégia de programação, vamos descrever, usando comentários, as variáveis que serão utilizadas, bem como seus respectivos tipos. Além disso, uma variável só pode ser referenciada se ele tiver sido inicializada, caso contrário, isso acarretará um erro no programa.

Após a descrição das variáveis (e constantes), temos que codificar a entrada e a saída do programa. A linguagem `Python` utiliza várias funções para a leitura da entrada. Inicialmente utilizaremos a função `input` para a leitura e a função `print` para impressão. Para utilizar essas funções não necessitamos incluir no programa nenhuma biblioteca.

Resolvido o problema da codificação da entrada e saída, falta computar o valor da receita. Isso pode ser feito da seguinte forma:

```
receita = taxaServico*instalacoes + custoMetro * metrosDeCabo
```

Vamos agora apresentar o programa completo. As fases 1 a 3 formam a parte da documentação do programa.

Para leitura de dados (fluxo de entrada), temos a função `input()` e para a impressão (fluxo de saída) temos a função `print()`.

```
1  # -*- coding: utf-8 -*-
2  # Programa: receita.py
3  # Programador:
4  # Data: 03/09/2017
5  # Este programa recebe como entrada o número de instalações e o preço da
6  # tarifa básica de instalação do ponto de TV a cabo, o total de metros de
7  # cabo coaxial utilizados nas instalações e o custo de cada metro de cabo
8  # instalado por um dado instalador de TV a cabo. Calcula a receita gerada
9  # pelo instalador e imprime o resultado.
10 # início do módulo principal
11 # descrição das variáveis locais utilizadas
12 # float tarifaServico, custoMetro, receita
13 # int   instalacoes, metrosDeCabo
14
15 # pré: tarifaServico custoMetro instalacoes metroDeCabo
16
17 # Passo 1. Leia a entrada
18 taxaServico = float(input('Entre com o valor da taxa de serviço R$: '))
19 custoMetro = float(input('Entre com o valor do metro do cabo R$: '))
20 instalacoes = int(input('Entre com o número de instalações: '))
21 metrosDeCabo = int(input('Entre com a quantidade de metros de cabo: '))
22 # Passo 2. Calcule a Receita
23 receita = taxaServico*instalacoes + custoMetro*metrosDeCabo
24 # Passo 3. Imprima a Receita
```

```
25 print('O valor da receita gerada foi: R$ {0:%.2f}'.format(receita))
26
27 # pós: receita == instalacoes * tarifaServico + custoMetro * metrosDeCabo
28 # fim do módulo principal
```

A primeira linha

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

indica que a codificação dos caracteres no programa será a UTF-8.

As linhas

```
custoMetro = float(input('Entre com o valor do metro do cabo R$: '))
instalacoes = int(input('Entre com o número de instalações: '))
```

ilustram as leituras de valores `float` e `int` pelo programa. Esse formato é utilizado pelo Python 3.7. A função `input()` lê um fluxo do dispositivo padrão de entrada e `int()` (ou `float()`) formata essa entrada como um inteiro (ou número de ponto flutuante).

A linha

```
receita = taxaServico*instalacoes + custoMetro*metrosDeCabo
```

busca na memória os valores armazenados nas variáveis `taxaServico`, `instalacoes`, `custoMetro` e `metrosDeCabo`, efetua as operações (na unidade lógica e aritmética), primeiro os produtos e depois a soma, começando da esquerda para a direita do símbolo de atribuição `=` e atribuiu o resultado no endereço de memória relativo a variável `receita`.

A linha

```
print('O valor da receita gerada foi: R$ {0:%.2f}'.format(receita))
```

ilustra a "impressão" no dispositivo padrão de saída da string

```
O valor da receita gerada foi: R$
```

seguido do valor da variável `receita` impresso com duas casas decimais (`%.2f`).

3 Exemplos de Problemas com Estrutura Sequencial

Nos exemplos desta seção as soluções dos problemas serão apresentadas seguindo todos os passos da metodologia proposta.

3.1 Divisão e Resto de Números inteiros

Vários problemas são solucionados usando o resto a divisão entre dois números inteiros. Vamos agora considerar o problema de ler dois números inteiros, calcular a divisão e o resto entre eles e imprimir o resultado.

Mesmo para um problema simples como esse, antes de continuar, mais informações a respeito do problema são necessárias para a construção do algoritmo que resolva esse problema. Necessitamos ter alguns cuidados na forma como o algoritmo irá ler os dados de entrada. A primeira é que temos que garantir que a entrada seja dada por dois números inteiros. A segunda é conhecer qual a ordem de entrada dos números, saber se o primeiro elemento a ser lido será o dividendo ou o divisor. Um outro ponto é conhecer qual será o intervalo de variação dos números. Além disso, temos que definir como será tratado o caso quando o divisor for igual a 0. Respondida essas questões, também temos que definir como será o formato da impressão dos resultados.

Após essas observações, nosso problema pode ser reescrito como: Dados dois números inteiros `dividendo`, `divisor`, projetar um algoritmo para computar a divisão e o resto entre esses números e imprimir os resultados usando o formato abaixo:

```
dividendo // divisor = quociente # divisão inteira
dividendo % divisor = resto
```

Vamos considerar o desenvolvimento de um programa chamado `divisaoresto` que computará o quociente e o resto de `dividendo` por `divisor`, onde `dividendo` e `divisor` são inteiros e o resultado da divisão (quociente) e o resto são números inteiros. Por exemplo, dados dois números inteiros, onde o `dividendo` é igual a 1265438 e o divisor é igual a 6543, utilizando instruções na linguagem `Python`, temos que o quociente da divisão inteira é dado por `1265438 // 6543 == 193` e o resto da divisão inteira é dado por `1265438 % 6543 == 2369`.

Veremos posteriormente que uma das características desejáveis num programa é a **robustez**. Por exemplo, temos que cuidar para que o programa funcione corretamente mesmo quando extrapolarmos os limites computacionais e matemáticos do programa. Por exemplo, precisamos nos preocupar com relação a casos não usuais, tais como quando `divisor == 0` e quais são os valores máximos e mínimos da entrada para `dividendo` e `divisor`. Nessa nossa primeira abordagem vamos assumir que o usuário não fornecerá como entrada o valor 0 para a variável `divisor` e que todos os valores da entrada podem ser representados pelo tipo `int` do Python.

Com essas observações, a descrição para esse problema fica bem mais simples:

```
# Este programa lê dois números inteiros dividendo e divisor, tal que
# o divisor != 0. Calcula o quociente e o resto e imprime o resultado.
```

Após a definição mais detalhada do problema, a próxima fase é o das especificações da entrada e saída. Nessa fase, a partir da definição mais detalhada do problema desenvolvemos especificações mais precisas a partir de um melhor entendimento do problema. Como cada "trecho" do programa pode ser visto como uma função, numa avaliação mais criteriosa, podemos especificar o domínio e a imagem para cada cada um desses "trechos". Mais precisamente, o domínio descreve o estado inicial em qualquer trecho do programa antes e sua execução (especificações de entrada - `pré:`) e a imagem descreve qual será o estado final após a execução desse trecho de programa (especificações de saída = `pós:`). Nesta disciplina vamos nos ater nas especificações de entrada e saída (pré e pós-condições) antes

do início e ao final do programa. Como vimos anteriormente, incluímos essas informações no programa como comentários e elas fazem parte da documentação e podem ser descritas da seguinte forma:

`# pré:` (para a entrada - *input*)

e

`# pós:` (para a saída - *output*)

Agora vamos identificar as `pré` (entrada) e `pós` (saída) condições para o problema da divisão e resto entre dois números inteiros.

Entrada	**Saída**
Dois números inteiros `dividendo`, `divisor` e `divisor != 0`	Quociente, Resto `quociente`, `resto`

Exemplo:

```
dividendo == 1265438 # entrada 1
divisor == 6543 # entrada 2

quociente = 1265438 // 6543 # cálculo do quociente inteiro no Python
quociente == 193 # valor da variável quociente armazenado na memória
resto = 1265438 % 6543 # cálculo do resto no Python
resto == 2639 # valor da variável resto armazenado na memória

# saída esperada
1265438 / 6543 = 193
1265438 % 6543 = 2639
```

Com isso podemos descrever as variáveis (iniciais) necessárias para a solução do problema:

```
# descrição das variáveis locais utilizadas
# int dividendo, divisor, quociente, resto
```

e as pré e pós-condições ficam:

```
# pré: dividendo divisor && divisor != 0

# pós: quociente, resto && quociente == dividendo //divisor &&
#      resto == dividendo % divisor
```

e nesse caso, usando as variáveis do tipo `int`, temos que as funções:

```
quociente(dividendo, divisor) = dividendo // divisor
resto(dividendo, divisor) = dividendo % divisor
```

transformam as pré condições que formam o domínio da função nas pós-condições que correspondem a imagem.

Uma vez definida a entrada (`pré:`) e a saída (`pós:`), a próxima fase é o desenvolvimento do algoritmo (refinamentos sucessivos). Nessa fase devemos descrever em passos elementares como o problema deve ser resolvido. Lembramos que um computador pode ler e imprimir dados, realizar operações aritmética e lógicas e controlar o fluxo dos passos (instruções).

Para o nosso exemplo temos que ler os dados, calcular a divisão e o resto e imprimir os resultados. Isso pode ser descrito da seguinte forma:

```
Algoritmo DivisaoResto
# O Este algoritmo lê dois números inteiros dividendo e divisor, tal que
# o divisor != 0. Calcula o quociente e o resto e imprime o resultado.
# descrição das variáveis locais utilizadas

# Passo 1. Leia dois números inteiros
# Passo 2. Calcule o quociente e o resto da divisão entre os dois números
# Passo 3. Imprima o quociente e o resto

# fim Algoritmo
```

O Passo 2 não tem como ser implementado diretamente num computador, uma vez que envolve o cálculo de dois valores. Nesses casos, fazemos um refinamento (subdivisão) do Passo 2, de tal forma que cada sub-passo possa ser descrito de acordo com as funcionalidades de um computador.

Podemos fazer o refinamento da seguinte forma:

```
# Passo 2. Calcule o quociente e o resto da divisão entre os dois números
# Passo 2.1. Calcule o quociente da divisão entre os dois números
# Passo 2.2. Calcule o resto da divisão entre os dois números
```

e com isso, a subdivisão fica:

```
Algoritmo DivisaoResto
# O Este algoritmo lê dois números inteiros, dividendo e divisor, tal que
# o divisor != 0. Calcula o quociente e o resto e imprime o resultado.
# descrição das variáveis locais utilizadas

# Passo 1. Leia dois números inteiros
# Passo 2. Calcule o quociente e o resto da divisão entre os dois números
# Passo 2.1. Calcule o quociente da divisão entre os dois números
# Passo 2.2. Calcule o resto da divisão entre os dois números
# Passo 3. Imprima o quociente e o resto

# fim Algoritmo
```

Vamos agora expressar a subdivisão do nosso problema de acordo com as funcionalidades de um computador e descrever os passos na linguagem Python. No caso da leitura dos dois números inteiros, necessitamos armazená-los na memória do computador. Para isso necessitamos identificá-los na memória, pois faremos operações com esses números. A forma de

identificá-los é com a utilização de identificadores. Como pretendemos que o nosso programa calcule a divisão e o resto entre dois números inteiros quaisquer, esses identificadores serão utilizados para representar variáveis do tipo inteiro.

Lembramos que num computador as informações estão armazenadas na memória. Isso é feito por meio da leitura de `dividendo` e `divisor`. Temos que definir e informar o usuário a ordem que os números inteiros representando o `dividendo` e o `divisor` serão lidos. O primeiro número inteiro a ser lido representará o dividendo e o segundo inteiro lido representará o divisor. Com essa decisão tomada, vamos agora traduzir os passos do nosso algoritmo usando a linguagem Python. Como vimos anteriormente, para ler dados do dispositivo de entrada padrão, podemos usar a função `input()`. Para converter a entrada lida num inteiro, usamos a função `int()`. A função `input()` também pode imprimir strings no dispositivo de saída padrão. A string que estiver dentro dos parêntesis da função `input('string')` será impressa no dispositivo de saída padrão. Podemos implementar o Passo 1 como:

```
# Passo 1. Leia dois números inteiros
dividendo = int(input('Leia o dividendo: '))
divisor = int(input('Leia o divisor: '))
```

A execução da instrução

```
dividendo = int(input('Leia o dividendo: '))
```

imprime na tela (ou dispositivo de saída padrão)

```
Leia o dividendo:
```

e interrompe a execução do programa e aguardando uma string do fluxo de entrada ser fornecida pelo usuário (`input()`). A string fornecida pelo usuário é convertida (desde que seja possível) para um número inteiro (`int()`) e atribuída à variável `dividendo`. De maneira análoga, um valor inteiro será armazenado na variável `divisor`. Após o Passo 1, os valores dos números inteiros `dividendo` e `divisor` estão armazenadas na memória.

O Passo 3 imprime os resultado usando um formato específico

```
dividendo / divisor = quociente
dividendo % divisor = resto
```

Como vimos anteriormente, podemos usar a função `print()` para imprimir uma string usando um formato específico.

```
# Passo 3. Imprima o quociente e resto
print('{0:d} / {1:d} = {2:d}').format(dividendo, divisor, quociente)
print('{0:d} % {1:d} = {2:d}').format(dividendo, divisor, resto)
```

No caso da impressão da string da primeira instrução `print()`

```
'{0:d} / {1:d} = {2:d}'
```

{0:d}, {1:d} e {2:d} serão substituído pelos valores das variáveis 0 (`dividendo`), 1 (`divisor`) e 2 (`quociente`), respectivamente, de `format(dividendo, divisor, quociente)`, e `d` indica que serão formatados como inteiros. De maneira análoga a impressão da string da segunda instrução `print`

```
'{0:d} / {1:d} = {2:d}'
```

substitui {0:d}, {1:d} e {2:d} pelos valores das variáveis 0 (`dividendo`), 1 (`divisor`) e 2 (`resto`), respectivamente, de `format(dividendo, divisor, resto)`.

Vamos agora descrever a implementação do Passo 2

```
# Passo 2. Calcule o quociente e o resto da divisão entre os dois números
# Passo 2.1. Calcule o quociente da divisão entre os dois números
# Passo 2.2. Calcule o resto da divisão entre os dois números
```

Utilizando a descrição das funções acima, podemos calcular e armazenar os resultados do quociente e do resto em variáveis inteiras específicas, os Passos 2.1 e 2.2 podem ser expressos da seguinte forma:

```
# Passo 2. Calcule o quociente e o resto da divisão entre os dois números
# Passo 2.1. Calcule o quociente da divisão entre os dois números
quociente =  dividendo // divisor
# Passo 2.2. Calcule o resto da divisão entre os dois números
resto = dividendo % divisor
```

Com essas observações, a codificação completa em Python 3.8 do nosso programa fica:

```
# -*- coding: utf-8 -*-
# Programa: divisaoresto.py
# Programador:
# Data: 12/10/2015
# O Este programa lê dois números inteiros dividendo e divisor, tal que
# o divisor != 0. Calcula o quociente e o resto e imprime o resultado.
# início do módulo principal
# descrição das variáveis locais utilizadas
# int dividendo, divisor, quociente, resto

# pré: dividendo divisor and divisor != 0

# Passo 1. Leia dois números inteiros
dividendo = int(input('Leia o dividendo: '))
divisor = int(input('Leia o divisor: '))
# Passo 2. Calcule o quociente e o resto da divisão entre os dois números
# Passo 2.1. Calcule o quociente da divisão entre os dois números
quociente =  dividendo // divisor
# Passo 2.2. Calcule o resto da divisão entre os dois números
resto = dividendo % divisor
# Passo 3. Imprima o quociente e resto
print('{0:d} / {1:d} = {2:d}').format(dividendo, divisor, quociente)
```

```
23 print('{0:d} % {1:d} = {2:d}').format(dividendo, divisor, resto)
24
25 # pós: quociente, resto and quociente == dividendo // divisor and
26 #      resto == dividendo % divisor
27 # fim do módulo principal
```

Vamos descrever alguns detalhes das principais características do `Python` no programa acima:

```
# Passo 1. Leia dois números inteiros
print('Leia dois números inteiros: ')
```

A função `print()` imprime a string que está entre `(' ')`. Como veremos posteriormente, a função `print` tem muitas outras funcionalidades.

Uma das formas de ler dados no `Python` é com a utilização da função `input()`. As duas instruções abaixo

```
# Passo 1. Leia dois números inteiros
....
dividendo = int(input('Leia o dividendo: '))
divisor = int(input('Leia o divisor: '))
```

imprimem as strings que estão entre `(' ')` e leem dois números inteiros e armazenam nas variáveis `dividendo` e `divisor`. A função `input()` lê uma string da entrada padrão (no nosso caso o teclado). Quando usamos `int(input())`, é feito a conversão da string para um número inteiro. Após a conversão os valores inteiros são atribuídos para as variáveis `dividendo` e `divisor`, respectivamente.

Essas instruções também poderiam ser usadas no seguinte formato:

```
# Passo 1. Leia dois números inteiros
.....
dividendo = int(input())
divisor = int(input())
```

O quociente de uma divisão inteira pode ser calculado em `Python` com a instrução

```
# Passo 2.1. Calcule o quociente da divisão entre os dois números
quociente = dividendo//divisor
```

onde o operador `'//'` computa o valor inteiro da divisão de `dividendo` por `divisor`. O resultado de `9 // 4` é igual a 2.

Para computar o resto de uma divisão inteiro usamos o operador `'%'`. O resultado de `9 % 4` é igual a 1.

Vamos descrever agora mais algumas características da função `print()`. A função `print` da instrução abaixo

```
# Passo 3. Imprima o quociente e resto
print('0:d / 1:d = 2:d'.format(dividendo, divisor, quociente))
```

imprime a string `'{0:d} / {1:d} = {2:d}'` usando o formado especificado, `{num:d}` (`num=0,1,2`), e as variáveis descritas em `format(...)`. No caso `{0:d}`, 0 indica que a primeira variável em `format()`, no caso `dividendo` será impresso nessa posição usando o formato de número inteiro `'d'`. Para `{1:d}` e `{2:d}` serão utilizadas as variáveis `divisor` e `quociente`, respectivamente, usando também o formato de número inteiro para serem "impressas" nessas posições. Para os valores de `dividendo == 9`, `divisor == 4`, o `quociente` será igual a 2 e na impressão haverá a "substituição" de `{0:d}` por 9, `{1:d}` por 4 e `{2:d}` por 2. A "impressão final fica:

```
9 / 4 = 2
```

Agora que descrevemos as características principais da linguagem `Python` utilizada para implementar o nosso programa vamos editá-lo e executá-lo. Usando o `IDLE` (ou uma outra IDE ou editor), edite o programa e salve-o num diretório de trabalho com o nome `divisaoresto.py`. Verifique se você está no diretório onde o programa fonte `divisaoresto.py` foi salvo e interprete/execute o programa. Você pode fazer isso diretamente no `IDLE` ou usando o comando:

```
python divisaoresto.py
```

Se você editou o programa corretamente a interpretação/execução do comando 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, interprete/execute novamente o programa.

```
# formato da entrada
Leia o dividendo: 9
Leia o divisor: 4

# formato da saída
9 / 4 = 2
9 % 4 = 1
```

Exemplo 2:

```
# formato da entrada
Leia o dividendo: 1265438
Leia o divisor: 6543

# formato da saída
1265438 / 6543 = 193
1265438 % 6543 = 2639
```

Após a edição e compilação (interpretação) sem erros do programa, a próxima fase é a 5, Teste e Verificação do Programa. Você pode executá-lo (de acordo com a linguagem e sistema operacional utilizado) e testar o seu funcionamento. Nesse nosso exemplo, como só vimos a estrutura de controle sequencial, assumimos que o usuário só fornecerá números que sejam válidos. De qualquer forma, teste o seu programa para números bem grandes e para divisor igual a 0 e verifique o que ocorre. Sugestões de testes:

(a) Teste o programa para vários valores de inteiros positivos.

(b) Teste o programa para $divisor = 0$.

(c) Teste o programa para $dividendo < 0$ e $divisor > 0$.

(d) Teste o Programa para $dividendo > 0$ e $divisor < 0$.

(e) Teste o Programa para $dividendo < 0$ e $divisor < 0$.

(f) Teste o programa para $dividendo = 4.2$ e $divisor = 2$.

(g) Teste o programa para $dividendo = 5$ e $divisor = 2.5$.

3.2 Perímetro e Área de um Círculo

Vamos agora resolver o problema de computar o perímetro e a área de um círculo de raio dado. Nosso programa deve ler o raio de um dado círculo, calcular e imprimir o perímetro e a área desse círculo. A entrada é dada por um número real positivo representando o raio de um círculo. A saída consiste em imprimir para o número real lido representando o valor do raio, o perímetro e a área do círculo, com mensagens apropriadas. O valor do raio, perímetro e área devem ser impressos com 5 espaços, sendo duas casas decimais. Como vimos anteriormente, o "." também ocupa um espaço e para obter esse formato de saída, dê uma lida no material sobre a função `print` e `format` da linguagem Python.

Considerando que o valor de entrada para o raio é um número positivo, o descrição para esse problema fica bem mais simples:

```
# Este programa lê um número to tipo float > 0 representando o valor do
# raio de um círculo, calcula o perímetro e a área do círculo e imprime o
# valor do raio do círculo, o perímetro e a área calculada.
```

Agora vamos identificar as `pré` (entrada) e `pós` (saída) condições para o problema de computar o perímetro e a área de um círculo.

Entrada	**Saída**
Um número do tipo `float` representando o `raio` do círculo e `raio > 0`	Perímetro e Área do círculo `perimetro`, `area`

```
Exemplo 1
# formato da entrada
2.0 # valor da entrada do raio do círculo

 # formato da saída
Raio =  2.00
Perímetro = 12.57
Área = 12.57
```

```
Exemplo 2
# formato da entrada
1.0 # valor da entrada do raio do círculo

# formato da saída
Raio =  1.00
Perímetro =  6.28
Área =  3.14
```

Com isso podemos descrever as variáveis iniciais necessárias para a solução do problema:

```
# descrição das constantes e variáveis locais utilizadas
# PI = 3.1415926536
# float raio, perimetro, area
```

e as pré e pós-condições ficam:

```
# pré: raio and raio > 0

# pós: perimetro, area and perimeto == 2*PI*raio and
#      area == PI*raio*raio
```

e nesse caso, usando as variáveis do tipo `float`, temos que as funções:

```
perimetro(raio) = 2*PI*raio
area(raio) = PI*raio*raio
```

transformam as pré condições que formam o domínio da função nas pós-condições que correspondem a imagem.

Uma vez definida a entrada (`pré:`) e a saída (`pós:`), a próxima fase é a construção do **algoritmo**. Nessa fase devemos descrever em passos elementares como o problema deve ser resolvido. Lembramos que um computador pode ler e imprimir dados, realizar operações aritmética e lógicas e controlar o fluxo dos passos (instruções).

O algoritmo abaixo implementa uma solução para esse problema.

```
# Algoritmo: perímetroareaC
# Este programa lê um número to tipo float > 0 representando o valor do
# raio de um círculo, calcula o perímetro e a área do círculo e imprime o
# valor do raio do círculo, o perímetro e a área calculada.
# início do algoritmo
# descrição das variáveis e constantes utilizados
# PI = 3.1415926536
# float raio, perimetro, area

# pré: raio && raio > 0

# Passo 1. Leia o valor do raio do círculo
```

```
# Passo 2. Compute o perímetro e a área do círculo
# Passo 3. Imprima o Perímetro e a Área

# pós: perimetro, area and perimeto == 2*PI*raio and
#      area == PI*raio*raio
# fim Algoritmo
```

O Passo 2 não tem como ser implementado diretamente num computador, uma vez que envolve o cálculo de dois valores. Nesses casos, fazemos uma subdivisão do Passo 2, de tal forma que cada sub-passo possa ser descrito de acordo com as funcionalidades de um computador.

Podemos fazer a subdivisão da seguinte forma:

```
# Passo 2. Compute o perímetro e a área do círculo
# Passo 2.1. Compute o perímetro do círculo
# Passo 2.2. Compute a área do círculo
```

e com isso, nosso algoritmo fica:

```
# Algoritmo: perímetroareaC
# Este programa lê um número to tipo float > 0 representando o valor do
# raio de um círculo, calcula o perímetro e a área do círculo e imprime o
# valor do raio do círculo, o perímetro e a área calculada.
# início do algoritmo
# descrição das variáveis e constantes utilizadas
PI = 3.1415926536
# float raio, perimetro, area

# pré: raio and raio > 0
# Passo 1. Leia o valor do raio do círculo
# Passo 2. Compute o perímetro e a área do círculo
# Passo 2.1. Compute o perímetro do círculo
# Passo 2.2. Compute a área do círculo
# Passo 3. Imprima o Perímetro e a Área

# pós: perimetro, area and perimeto == 2*PI*raio and
#      area == PI*raio*raio
# fim do algoritmo
```

Vamos agora expressar a subdivisão do nosso problema de acordo com as funcionalidades de um computador e descrever os passos na linguagem Python.

```
# Passo 1. Leia o valor do raio do círculo
print('Entre com o valor do raio: ')
raio = float(input())
```

neste caso, estamos usando a função `print()` para orientar a entrada do usuário. Como no exemplo anterior, poderíamos usar a mensagem com a função `input()`.

```
# Passo 1. Leia o valor do raio do círculo
raio = float(input('Entre com o valor do raio: '))
```

O Passo 3 imprime os resultado usando um formato específico

```
Raio =  1.00
Perímetro =  6.28
Área =  3.14
```

Como vimos anteriormente, podemos usar a função `print()` para imprimir uma string usando um formato específico.

```
# Passo 3. Imprima o Perímetro e a Área
print('Raio = {0:5.2f}'.format(raio))
print('Perímetro = {0:5.2f}'.format(perimetro))
print('Área = {0:5.2f}'.format(area))
```

neste caso, `{0.5.2f}` indica que o valor da variável será impresso no dispositivo padrão de saída usando 5 espaços, alinhado pela direita, sendo 2 espaços para a parte inteira (se o número da parte inteira tiver menos que dois caracteres, serão usados caracteres espaços à direita do número da parte inteira para completar os dois caracteres e se a parte inteira tiver mais que dois caracteres, todos os caracteres serão impressos, da esquerda para a direita), um espaço para o caractere "." e dois caracteres para a parte decimal, No caso da parte decimal, é feita uma aproximação, por exemplo, a parte decimal 155 será impressa como 16.

Vamos agora descrever a implementação do Passo 2

```
# Passo 2. Compute o perímetro e a área do círculo
# Passo 2.1. Compute o perímetro do círculo
# Passo 2.2. Compute a área do círculo
```

Utilizando a descrição das funções acima, podemos calcular e armazenar os resultados do perímetro e do raio em variáveis do tipo `float` específicas, os Passos 2.1 e 2.2 podem ser expressos da seguinte forma:

```
# Passo 2. Compute o perímetro e a área do círculo
# Passo 2.1. Compute o perímetro do círculo
perimetro = 2 * PI * raio
# Passo 2.2. Compute a área do círculo
area = PI * raio*raio
```

No caso da constante `PI` (π), como estamos usando o Python, podemos usar a constante existente na biblioteca/módulo `math`. Para isso é necessário importar o módulo (`import mat`) e usar `math.pi` como a constante para o valor de `PI` (π).

```
# declaração dos módulos utilizados
import math
```

com isso, podemos reescrever o Passo 2.1 da seguinte maneira:

```
# Passo 2.1. Compute o perímetro do círculo
perimetro = 2 * math.pi * raio
```

Com relação ao Passo 2.2., para calcular o quadrado do `raio` também podemos usar o operador `**` ou usar a função/método `pow(base, expoente)` da biblioteca/módulo `math`. com isso, podemos reescrever o Passo 2.2 das seguintes formas:

```
# Passo 2.2. Compute a área do círculo
area = math.pi * raio**2
```

ou

```
# Passo 2.2. Compute a área do círculo
area = math.pi * math.pow(raio,2)
```

Com isso o programa fica:

```
1  # -*- coding: utf-8 -*-
2  # Programa: perimetroareaC.py
3  # Programador:
4  # Data: 04/04/2012
5  # Este programa lê um número to tipo float > 0 representando o valor do
6  # raio de um círculo, calcula o perímetro e a área do círculo e imprime o
7  # valor do raio do círculo, o perímetro e a área calculada.
8  # descrição dos módulos/bibliotecas utilizados
9  import math
10 # início do módulo principal
11 # descrição das variáveis utilizadas
12 # float raio, perimetro, area
13
14 # pré: raio and raio > 0
15
16 # Passo 1. Leia o valor do raio do círculo
17 print('Entre com o valor do raio: ')
18 raio = float(input())
19 # Passo 2. Compute o perímetro e a área do círculo
20 # Passo 2.1. Compute o perímetro do círculo
21 perimetro = 2 * math.pi * raio
22 # Passo 2.2. Compute a área do círculo
23 area = math.pi * raio**2
24 # Passo 3. Imprima o Perímetro e a Área
25 print('Raio = {0:5.2f}'.format(raio))
26 print('Perímetro = {0:5.2f}'.format(perimetro))
27 print('Área = {0:5.2f}'.format(area))
28
29 # pós: perimetro, area and perimeto == 2*PI*raio and
30 #      area == PI*raio*raio
31 # fim do módulo principal
```

Você pode editar o programa usando um editor ou o `IDLE`.

Após a edição e compilação (interpretação) sem erros do programa, a próxima fase é o 5, Teste e Verificação do Programa. Você pode executá-lo (de acordo com o sistema operacional utilizado) e testar o seu funcionamento. Nesse nosso exemplo, como só vimos a estrutura de controle sequencial, assumimos que o usuário só fornecerá números que sejam válidos. De qualquer forma, teste o seu programa para números bem grandes e para raio igual ou menor que 0 e verifique o que ocorre. Abaixo, um exemplo de uma execução do programa no ambiente Linux.

```
$ python perimetroareaC.py
```

Teste o seu programa para vários valores e verifique se o programa está computando corretamente os valores de perímetro e área.

```
Exemplo 1
# formato da entrada
1.0

# formato da saída
Raio =  1.00
Perímetro =  6.28
Área =  3.14
```

```
Exemplo 2
# formato da entrada
2.0

# formato da saída
Raio =  2.00
Perímetro = 12.57
Área = 12.57
```

3.3 Concatenando Palavras

Vamos ampliar a definição da aula anterior de uma palavra. Vamos considerar uma **palavra** como sendo um conjunto de caracteres $c_0c_2 \ldots c_{k-1}$, onde cada c_i é um é um caractere visível do conjunto ASCII. Toda palavra é precedida de um caractere espaço e após a palavra também tem um caractere espaço.

Vamos agora resolver o problema de concatenar duas palavras. Considerando a definição de palavra acima, podemos descrever esse problema da seguinte forma:

```
# Este algoritmo lê duas palavras e computa a concatenação das duas
# palavras e imprime palavra concatenada.
```

Agora vamos identificar as `pré` (entrada) e `pós` (saída) condições para o problema da concatenação de duas palavras.

Entrada	**Saída**
Duas palavras formadas por caracteres visíveis `char[i]` tal que `palavra1 == char[0]..char[n-1]` e `palavra2 == char[0]..char[m-1]`, onde `n == len(palavra1)` e `m == len(palavra2)`	Uma palavra resultante da concatenação de `palavra1` e `palavra2` tal que `palavra == char[0]..char[n-1]char[0]..char[m-1]` e `n + m == len(palavra)`

```
Exemplo 1
# formato da entrada
primeiro
segundo

# formato da saída
primeirosegundo
```

```
Exemplo 2
# formato da entrada
aula
pratica

# formato da saída
aulapratica
```

Com isso podemos descrever as variáveis necessárias para a solução do problema:

```
# descrição das variáveis locais utilizadas
# str palavra1, palavra2, palavra
```

e as pré e pós-condições ficam:

```
# pré: palavra1 palavra2

# pós: palavra and palavra == palavra1 + palavra2
```

e nesse caso, usando as variáveis do tipo `str`, temos que a função:

```
concatena(palavra1,palavra2) = palavra1 + palavra2
```

transformam as pré condições que formam o domínio da função nas pós-condições que correspondem a imagem.

Uma vez definida a entrada (`pré:`) e a saída (`pós:`), a próxima fase é projetarmos o **algoritmo**. Nessa fase devemos descrever em passos elementares como o problema deve ser resolvido. Lembramos que um computador pode ler e imprimir dados, realizar operações aritmética e lógicas e controlar o fluxo dos passos (instruções).

O algoritmo abaixo implementa uma solução para esse problema.

```
# Algoritmo: ConcatenaPalavras
# Este algoritmo lê duas palavras e computa a concatenação das duas
# palavras e imprime palavra concatenada.
# início do algoritmo
# descrição das variáveis utilizadas
# str palavra1, palavra2, palavra

# pré: UmChar[1]...UmChar[n] == palavra1 and
#      UmChar[1]...UmChar[m] == palavra2

# Passo 1. Leia duas palavras
# Passo 2. Compute a concatenação das duas palavras
# Passo 3. Imprima a concatenação das duas palavras

# pós: palavra and palavra == palavra1 + palavra2
# fim do algoritmo
```

Vamos agora expressar o algoritmo do nosso problema de acordo com as funcionalidades de um computador e descrever os passos na linguagem Python.

```
# Passo 1. Leia duas palavras
print('Leia a palavra1: ')
palavra1 = input()
print('Leia a palavra2: ')
palavra2 = input()
```

neste caso, estamos usando a função `print()` para orientar as entradas do usuário. Como vimos anteriormente, o fluxo de entrada já é dado por uma string de caracteres, com isso não é necessário converter as entradas obtidas pela função `input()`. A forma como projetamos a entrada do Passo 1 requer que as entradas de `palavra1` e `palavra2` sejam feitas em linhas separadas. O Python tem uma função para ler mais de uma variável na mesma linha. Isso pode ser feito com as funções `map()` e `split()` da seguinte forma:

```
# Passo 1. Leia as duas palavras
palavra1, palavra2 = map(str, input(),split())
```

e com isso, a entrada pode ser dada como:

```
Exemplo 1
primeiro segundo
```

O Passo 3 imprime os resultado usando um formato específico

```
primeiro concatenada com segundo == primeirosegundo
```

Como vimos anteriormente, podemos usar a função `print()` com `format()` para imprimir uma string usando um formato específico.

```
# Passo 3. Imprima a concatenação das duas palavras
print('{0} concatenada com {1} == {2}').format(palavra1, palavra2, palavra))
```

sendo que na impressão sem formatação não é necessário especificar o tipo que está sendo impresso, basta a posição .

Vamos agora descrever a implementação do Passo 2

```
# Passo 2. Compute a concatenação das duas palavras
```

Utilizando a descrição da função acima, podemos calcular e armazenar os resultados da concatenação das duas palavras numa variável do tipo `str` específica, o Passo 2 pode ser expresso da seguinte forma:

```
# Passo 2. Compute a concatenação das duas palavras
palavra = palavra1 + palavra2
```

Abaixo descrevemos o programa `concatena.py` que implementa uma solução para o problema da concatenação de duas palavras.

```
1  # -*- coding: utf-8 -*-
2  # Programa: concatena.py
3  # Programador:
4  # Data: 16/04/2013
5  # Este algoritmo lê duas palavras e computa a concatenação das duas
6  # palavras e imprime palavra concatenada.
7  # inicio do módulo principal
8  # descrição das variáveis locais utilizadas
9  # str palavra1, palavra2, palavra
10
11 # pré: palavra1 palavra2
12
13 # Passo 1. Leia duas palavras
14 print('Leia a palavra1: ')
15 palavra1 = input()
16 print('Leia a palavra2: ')
17 palavra2 = input()
18 # Passo 2. calcule a concatenação das duas palavras
19 palavra =  palavra1 + palavra2
20 # Passo 3. Imprima as concatenação das duas palavras
21 print('{0} concatenada com {1} == {2}').format(palavra1,palavra2,palavra))
22
23 # pós: palavra and palavra == palavra1 + palavra2
24 # fim do módulo principal
```

Usando o IDLE ou um editor, edite o programa cuidadosamente e execute-o usando a opção no IDLE ou inicialize um terminal e execute

```
$ python concatena.py
```

se você estiver usando um editor. Caso a interpretação do programa gere alguma advertência ou erro, veja o número da linha e verifique o que você digitou de forma errada. Após ter corrigido as advertências e erros, execute novamente o programa.

Exemplos de execução:

```
# formato da entrada
primeiro
segundo

# formato da saída
primeiro concatenada com segundo == primeirosegundo
```

```
# formato da entrada
aula
pratica

# formato da saída
aula concatenada com pratica == aulapratica
```

3.4 Pontos e Linhas

Vamos agora ilustrar a solução de um problema que utiliza recursos gráficos. Como no exemplo da Aula 3, vamos utilizar a biblioteca `pillow` (`PIL`) para solucionar o problema de dados duas coordenadas (x_1, y_1) e (x_2, y_2) computar e exibir a reta ligando os dois pontos.

A biblioteca possui vários módulos e funções que auxiliam e simplificam a manipulação dos recursos gráficos do Python. Neste programa vamos descrever a utilização dos funções/métodos dos módulos `Image` e `ImageDraw`. Para gerar a imagem, inicialmente temos que criar a imagem e criar um "ambiente" de desenho. Depois com o uso da função/método `line([(x1,y1), (x2,y2)])` criamos um conjunto pixels ligando `(x1,y1)` a `(x2,y2)`) na imagem criada (inicialmente vazia). Usando a função/método `show` no módulo `Image` "imprimimos" a imagem gerada.

```
1  # -*- coding: utf-8 -*-
2  # Programa: retapil.py
3  # Programador:
4  # Data: 20/08/2020
5  # Este programa utiliza o módulo PIL (pillow). O programa lê duas
6  # coordenadas (x1,y1) e (x2,y2) e gera e imprime uma reta entre os pontos
7  # Declaração dos módulos utilizados
8  from PIL import Image, ImageDraw
9  # início do módulo principal
10 # descrição das variáveis utilizadas
11 # Image imagem
12 # ImageDraw desenhe
13 # int largura, altura, x1, y1, x2, y2
14 # str padrao
15
16 # pré: x1 y1 x2 y2
```

```
17 # Passo 1. Defina o ambiente gráfico e leia os pontos
18 # Passo 1.1. Defina o tamanho da imagem)
19 largura = 1200
20 altura = 800
21 # Passo 1.2. Defina o padrão de cores
22 padrao = 'RGB'
23 # Passo 1.3. Crie a imagem
24 imagem = Image.new(padrao, (largura,altura), color='white')
25 desenhe = ImageDraw.Draw(imagem)
26 # Passo 1.4 Leia duas coordenadas (x,y)s
27 x1,y1 = map(int,input().split())
28 x2,y2 = map(int,input().split())
29 # Passo 2. Gere uma reta vermelha (x1,y1)-(x2,y2) na imagem
30 desenhe.line([(x1,y1),(x2,y2)],fill=(255,0,0))
31 # Passo 3. Mostre a reta na imagem
32 imagem.show()
33
34 # pós: uma reta (x1,y1)-9x2,y2) na imagem
35 # fim do módulo principal
```

Usando o IDLE ou um editor, edite o programa cuidadosamente e execute-o usando a opção no IDLE ou inicialize um terminal e execute

```
$ python retapil.py
```

se você estiver usando um editor. Caso a interpretação do programa gere alguma advertência ou erro, veja o número da linha e verifique o que você digitou de forma errada. Após ter corrigido as advertências e erros, execute novamente o programa.

```
# formato da entrada
120 1080
80 720
```

A instrução

```
# Passo 1.2. Defina o padrão de cores
padrao = 'RGB'
```

define o sistema de cores que será utilizado. As instruções

```
# Passo 1.3. Crie a imagem
imagem = Image.new(padrao, (largura,altura), color='white')
desenhe = ImageDraw.Draw(imagem)
```

gera uma imagem vazia com fundo branco de tamanho 1200 × 800 pixels. A variável `desenhe` recebe um "contexto" para executar operações na imagem. A instrução

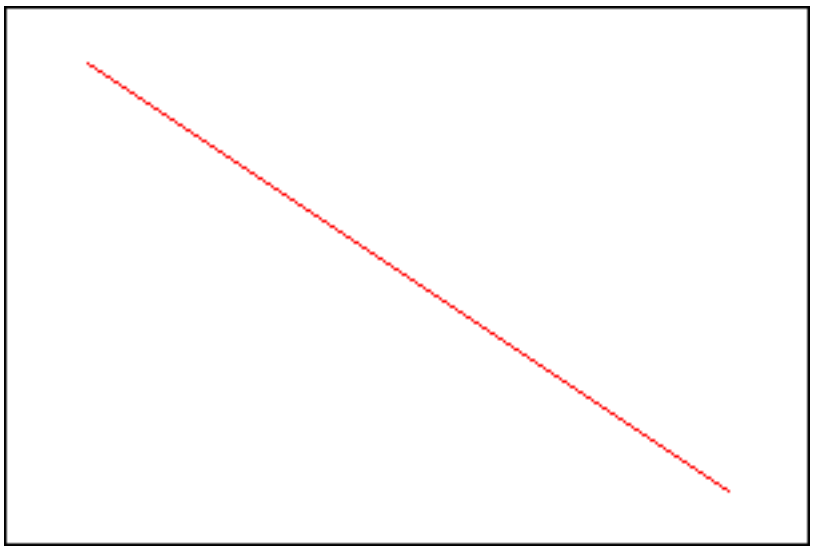

Figura 1: Saída do Programa `retapil.py`.

```
# Passo 2. Gere uma reta vermelha (x1,y1)-(x2,y2) na imagem
desenhe.line([(x1,y1),(x2,y2)],fill=(255,0,0))
```

gera uma linha entre as coordenadas (x_1, y_1) e (x_2, y_2). Finalmente, a instrução

```
# Passo 3. Mostre a reta na imagem
imagem.show()
```

mostra a imagem com uma linha ligando (x_1, y_1) a (x_2, y_2).

Esse mesmo problema pode ser resolvido com o uso da biblioteca `turtle`. A biblioteca também possui funções/métodos para gerar objetos gráficos numa "imagem". Caso utilizemos os tamanhos padrões, não é necessário criar a "tela" (imagem) para gerar os gráficos. Da mesma forma, não temos que mostrar ("imprimir") a imagem.

```
1  # -*- coding: utf-8 -*-
2  # Programa: retaturtle.py
3  # Programador:
4  # Data: 21/08/2020
5  # Este programa utiliza a biblioteca turtle.
6  # O programa lê dois pontos (x,y) e gera e imprime uma reta
7  # entre os dois pontos.
8  # Declaração dos módulos utilizadas
9  from turtle import *
10 # início do módulo principal
11 # descrição das variáveis utilizadas
12 # float x1, y1, x2, y2
13
14 # pré: x1 y1 x2 y2
15
16 # Passo 1. Leia a entrada e defina a as variáveis
17 # Passo 1.1. Leia duas coordenadas (x,y)
18 x1, y1 = map(float,input().split())
```

```
19 x2, y2 = map(float,input().split())
20 # Passo 1.2. Defina a cor
21 colormode(255) # escala [0,255]
22 pencolor(255,0,0) # cor vermelha
23 # Passo 2. Gere e imprima a reta entre os dois pontos
24 # Passo 2.1. Apague a tartaruga
25 hideturtle()
26 # Passo 2.2. Levante a caneta
27 penup()
28 # Passo 2.3. Vá para a posição inicial
29 setpos(x1,y1)
30 # Passo 2.4. Abaixe a caneta
31 pendown()
32 # Passo 2.3. Vá para a posição final
33 setpos(x2,y2)
34
35 # pós: uma reta vermelha ligando dois pontos
36 # fim do módulo principal
```

Usando o IDLE ou um editor, edite o programa cuidadosamente e execute-o usando a opção no IDLE ou inicialize um terminal e execute

```
$ python retaturtle.py
```

se você estiver usando um editor. Caso a interpretação do programa gere alguma advertência ou erro, veja o número da linha e verifique o que você digitou de forma errada. Após ter corrigido as advertências e erros, execute novamente o programa.

```
# formato da entrada
-200 -200
200 200
```

O ambiente `turtle` sempre incia com a tartaruga (turtle) no meio da tela, posição $(0, 0)$. A escala RGB também padrão é no intervalo $[0.0, 1.0]$. A tartaruga também sempre é visível por padrão. O desenhos são os "rastro" da tartaruga. O desenho é "feito" por uma caneta (`pen`).

As instruções

```
# Passo 1.2. Defina a cor
colormode(255) # escala [0,255]
pencolor(255,0,0) # cor vermelha
```

altera a escala de cores para $[0, 255]$ e define que a caneta usará a cor vermelha. As instruções

```
# Passo 2.1. Apague a tartaruga
hideturtle()
# Passo 2.2. Levante a caneta
penup()
```

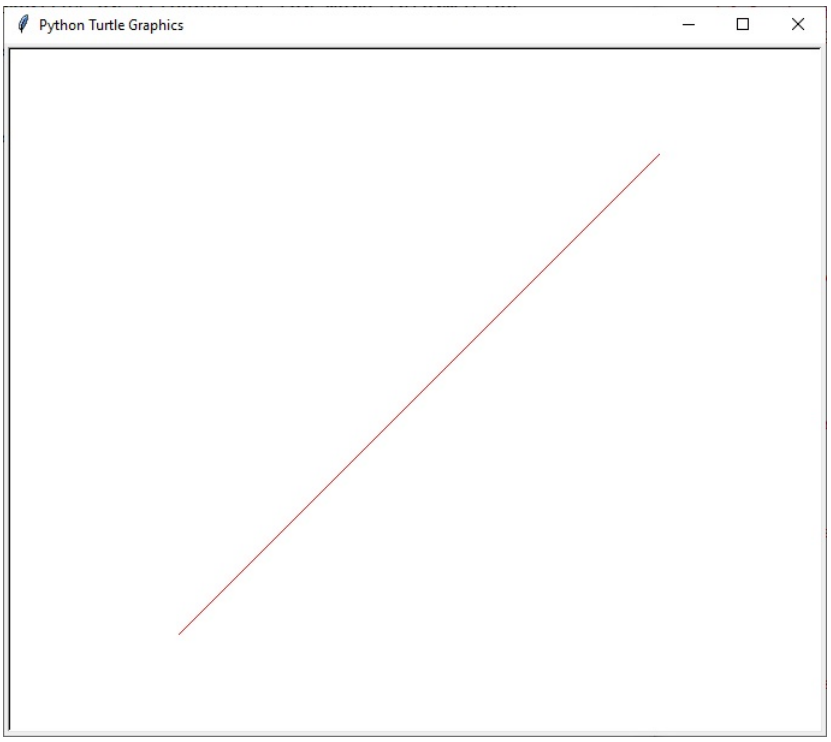

Figura 2: Saída do Programa `retaturtle.py`.

torna a tartaruga invisível e levanta a "pena" da caneta, pois o ambiente sempre incia na posição$(0, 0)$ e se não levantarmos a caneta, qualquer movimento gerara um rastro de $(0, 0)$ até o ponto onde começara ser gerado qualquer desenho. A instrução

```
# Passo 2.3. Vá para a posição inicial
setpos(x1,y1)
```

move a tartaruga para a posição `(x1,y1)`. A instrução

```
# Passo 2.4. Abaixe a caneta
pendown()
```

abaixa a "pena" da caneta. A partir de agora, qualquer movimentação da tartaruga gerará um "rastro". A instrução

```
# Passo 2.3. Vá para a posição final
setpos(x2,y2)
```

move a tartaruga para a posição `(x2,y2)`. Como a "pena" da caneta está abaixada, a movimentação da tartaruga gerará um rastro.

Utilize a sua criatividade para gerar outras figuras, mudar as cores e explorar as bibliotecas `PIL` (`pillow`) e `turtle`. A execução do programa `relapil.py`necessita que a biblioteca `pillow` esteja instalada. No caso do programa `retaturtle.py` não é necessária nenhuma instalação adicional, pois a biblioteca `turtle` já vem instalada. Nas próximas aulas abordaremos outras funcionalidades dessas bibliotecas.